THE FORTRAN 77 REFERENCE GUIDE IDR4029

This guide documents the operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 17 (Rev. 17).

PRIME Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

ACKNOWLEDGEMENTS

We wish to thank the members of the documentation team and also the non-team members, both customer and Prime, who contributed to and reviewed this book.

First Printing January 1980

All correspondence on suggested changes to this document should be directed to:

John Mann
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

CONTENTS

SECTION 1

INTRODUCTION

DEFINITIONS

There are many versions of FORTRAN. The following names are used for them in this guide:

FORTRAN: A mathematically oriented programming language developed by IBM in the 1950's.

FORTRAN 66: A standardized FORTRAN, defined in the American National Standards Institute (ANSI) publication "ANSI X3.9-1966".

FORTRAN IV: Any version of FORTRAN which is based on ANSI X3.9-1966 and contains extensions developed by a particular computer manufacturer.

FTN: Prime FORTRAN IV

FORTRAN 77: A new standardized FORTRAN, defined in the American National Standards Institute publication "ANSI X3.9-1978".

F77: Prime's extended version of FORTRAN 77. The F77 language conforms fully to ANSI X3.9-1978.

Certain FORTRAN-specific terms used in this introduction are formally defined at the beginning of Section 2.

THIS DOCUMENT

This document is a programmer's guide to the FORTRAN 77 language as implemented on the Prime system. The reader is expected to be familiar with some version of FORTRAN, and with programming in general, but not necessarily with Prime computers. A one-semester course in FORTRAN programming should provide sufficient background.

Users familiar with programming but not with FORTRAN should consult an appropriate FORTRAN 77 textbook. Some examples are:

Katzan, Harry, FORTRAN 77, Van Nostrand Reinhold Company, New York, 1979

Wagener, Jerrold L., Principles of FORTRAN 77 Programming, John Wiley and Sons, New York, 1980

This document contains the following:

- An introduction to the F77 language.
- All the information from ANSI X3.9-1978 which a programmer needs to program in FORTRAN 77. Various details elaborated in the standard for the sake of completeness, but unlikely ever to be required in practice, have been omitted to limit this guide to a reasonable size.
- Complete information on all Prime extensions to FORTRAN 77.
- Complete information on the use of the F77 compiler
- Suggestions for optimizing F77 programs.
- An appendix on converting programs from FTN to F77.
- An appendix containing an F77 program example demonstrating the more significant features of the language.
- Appendices detailing the ASCII character set and the storage formats used for the F77 data types.

RELATED DOCUMENTS

The following documents contain additional information relevant to programming in F77.

The Prime User's Guide

Nearly all the information in The FORTRAN 77 Reference Guide (this guide) relates directly to F77. Little general information about using the Prime computer system is presented here.

Complete instructions for creating, loading, and executing programs in Prime FORTRAN 77 or any Prime language, plus extensive additional information on Prime system utilities for programmers, is found in The Prime User's Guide. The user's guide and this reference guide are complementary documents: both are essential to the F77 programmer.

The User's Guide also contains a complete description of all Prime documents.

The FORTRAN IV Reference Guide

The FTN language is described in The FORTRAN IV Reference Guide, FDR3Ø57. Those involved with converting programs from FTN to F77 should have a copy of that guide, since it contains some information that applies to FTN but not F77. Such information is not reiterated in this guide. See Appendix A for information on the conversion of FTN

programs to F77.

The ANSI Standard

The definitive reference for FORTRAN 77 is "ANSI X3.9-1978 Programming Language FORTRAN". Every installation which uses FORTRAN 77 extensively should have a copy of this standard, which may be obtained from American National Standards Institute, 1430 Broadway, New York, NY, 10018.

FORTRAN 77

In 1978, ANSI published "ANSI X3.9-1978 Programming Language FORTRAN." This standard exhaustively defines a new version of FORTRAN, called FORTRAN 77. The new FORTRAN includes and standardizes nearly all the useful extensions to FORTRAN 66 developed by individual manufacturers. The result is a comprehensive, well-defined, and powerful language.

Development of FORTRAN 77 continues at the ANSI level.

NEW FEATURES IN FORTRAN 77

FORTRAN 77 provides many capabilities additional to those of FORTRAN 66. Some of them have been used in nearly all manufacturers' versions of FORTRAN IV, but have not previously been defined in any standard. Many of them were incorporated into FTN on the basis of preliminary documents released by ANSI, to facilitate the eventual conversion of FTN programs to F77.

The features available in FORTRAN 77 but not in FORTRAN 66 are as follows:

Data Declaration Capabilities

- A statement to name the main program (PROGRAM statement)
- An implicit type-rule for default typing of data items by first letter (IMPLICIT statement)
- Named constants (PARAMETER statement)
- A CHARACTER data type
- Arrays with up to seven dimensions
- Explicit lower bounds for array dimensions
- Array bounds with positive, zero, or negative values
- Integer constant expressions in array-bound specifications

Execution-Time Capabilities

- Operations to concatenate and extract substrings from CHARACTER data
- Use of an array name, character substring, or implied-DO list in a DATA statement
- Use of integer expressions (rather than just integers) for array subscripts, selection values for computed GO TO's, and file units referred to in BACKSPACE, ENDFILE, and REWIND statements
- Use of integer, real, or double precision expressions for DO-loop and implied-DO index and control values
- DO and implied-DO loops that may execute zero times and have negative incrementation values
- A block-IF statement, with subsidiary ELSE IF, ELSE, and END IF statements, for conditional execution of blocks of statements
- Use of a format statement label in an ASSIGN statement
- Use of decimal digits or a character string in a PAUSE or STOP statement

Subprogram Capabilities

- Multiple entry points to subprograms
- Alternate returns in subroutines
- Differentiation between external (user-supplied) and intrinsic (built-in) functions
- Generic names for intrinsic functions
- Functions with no arguments
- More than one block data subprogram

Input/Output Capabilities

- Direct-Access Files
- List-Directed I/O
- Internal (storage-to-storage) formatted data transfer
- Statements to open and close files, and to inquire about the

status of a file

- Additional edit-control descriptors for formatted I/O, such as sign control, blank editing, and tabbing

PRIME EXTENSIONS TO FORTRAN 77

Unextended FORTRAN 77 already includes features to perform nearly every programming task for which the FORTRAN language is appropriate. Prime has avoided extending its FORTRAN 77 unnecessarily, since needless extensions would serve mostly to reduce compatibility between F77 and other versions of FORTRAN 77.

Prime has extended its FORTRAN 77 for three reasons:

- To provide added power and convenience of use to the language
- To take advantage of particular features of the Prime computer system
- To provide the maximum possible compatibility with FTN, and substantial compatibility with IBM and other manufacturers' versions of FORTRAN IV. See Appendix A for information on the conversion of FTN programs to F77.

Some extensions belong in two or all three categories.

The extensions of greatest interest to a new F77 user are listed below. All F77 extensions are described in detail at appropriate places later in this guide.

- Variable and array names may have up to 32 characters, may contain lowercase letters, and may contain the characters "$" and "_".
- Comments may appear anywhere in a statement. Blank lines may appear in a program unit; they are treated as comments.
- Various synonyms for the FORTRAN data types are provided. COMPLEX*16, INTEGER*2, LOGICAL*2, and LOGICAL*1 data types have been added.
- Extended intrinsics to deal with the extended data types are provided.
- Octal constants are accepted in F77 source text.
- Data may be initialized in a type-declaration statement.
- CHARACTER and non-CHARACTER data may be equivalenced.
- All COMMON block data is static. Blank COMMON may be initialized.

- IBM syntax for direct-access READs and WRITEs is accepted.
- Recursion is permitted in subroutines, though not in functions.
- The B field descriptor for formatting business data (similar to PICTURE formatting in COBOL and PL/I) is provided.
- Files can be automatically inserted into the source file by the compiler.

There are various other extensions to allow certain FTN constructs that are not standard in FORTRAN 77 to be accepted by the F77 compiler. These need not be enumerated here. They are described in Appendix A.

F77 RESTRICTIONS

The segmented nature of the Prime virtual memory system imposes a few restrictions on F77 programs. None of them are contrary to the ANSI standard or need interfere with program design.

- The executable code (exclusive of data storage) for a program unit may not occupy more than one segment (128K bytes)
- No program unit may have more than one segment of local static storage. (For additional static storage, move some of the data to a COMMON block.)
- No program unit may have more than one segment of dynamic storage. (Make the excess static.)
- No data item in a COMMON block may be split across the boundary between two segments. Methods for complying with this rule are described under COMMON Statement in Section 3.

INTERFACE TO OTHER LANGUAGES

Since all Prime high-level languages are alike at the object-code level, and since all use the same calling conventions, object modules produced by the F77 compiler can reference and be referenced by modules produced by the FTN, COBOL, or PL1G compilers, provided that certain restrictions are observed:

- All I/O routines must be written in the same language
- There must be no conflict of data types for variables being passed as arguments. For example, an INTEGER in FORTRAN 77 should be declared as FIXED BINARY in PL/I. See Appendix C for a description of F77 data storage formats.
- Modules compiled in 64V or 32I mode cannot reference or be referenced by modules compiled in any R mode. Modules in 64V or 32I may reference each-other if they are otherwise compatible.

A few special restrictions apply when F77 and FTN modules reference each-other. These are discussed in Appendix A.

F77 program units can also reference PMA (Prime Macro Assembler) routines, and vice versa. For information, see The Assembly Language Programmer's Guide.

F77 AND PRIME UTILITIES

Prime offers three major utility systems for use by Prime programmers. These are:

- Multiple Index Data Access System (MIDAS)
- Forms Management System (FORMS)
- Database Management System (DBMS)

For complete information on any of these utilities, see the appropriate reference guide. Following is a brief description of MIDAS and FORMS. At initial release, F77 does not provide an interface with DBMS.

Multiple Index Data Access System (MIDAS)

MIDAS is a system of interactive utilities and high-level subroutines enabling the use of index-sequential and direct-access data files at the applications level. Handling of indices, keys, pointers, and the rest of the file infra-structure is performed automatically for the user by MIDAS. Major advantages of MIDAS are:

- Large data files may be constructed
- Efficient search techniques
- Rapid data access
- Compatibility with existing Prime file structures
- Ease of building files
- Primary key and up to 19 secondary keys possible
- Multiple user access to files
- Data entry lockout protection
- Partial/full file deletion utility

The interface of F77 with MIDAS is identical to that of FTN.

See: Reference Guide, Multiple Index Data Access System (MIDAS)

Forms Management System (FORMS)

The Prime Forms Management System (FORMS) provides a convenient method of defining a form in a language specifically designed for such a purpose. These forms may then be implemented by any applications program which uses Prime's Input/Output Control System (IOCS), including programs written in F77. Applications programs communicate with FORMS through input/output statements native to the host language. Programs that currently run in an interactive mode can easily be converted to use FORMS.

FORMS allows cataloging and maintenance of form definitions available within the computer system. To facilitate use within an applications program, all form definitions reside within a centralized directory in the system. This directory, under control of the system administrator, may be easily changed, allowing the addition, modification, or deletion of form definitions.

The interface of F77 with FORMS is identical to that of FTN.

See: FORMS Management System

THE SOURCE LEVEL DEBUGGER

Prime makes available a powerful interactive debugging tool, the Source Level Debugger, which may be obtained by any Prime installation as a separately priced item. Use of the debugger can greatly expedite and simplify the debugging process. Major features of the debugger enable the programmer to:

- Set both absolute and conditional breakpoints
- Request the execution of debugger commands (action list) when a breakpoint occurs
- Execute the program step by step
- Call subroutines or functions from debugger command level
- Trace statement execution
- Trace selected variables, printing a message when their value changes
- Print and/or change the value of any variable
- Print a subprogram call/return stack history (traceback)
- Examine the source file while executing within the debugger, eliminating the need for hard-copy listings

See: The Source Level Debugger Reference Guide.

THE CONDITION HANDLING MECHANISM

When an error occurs during execution of a program, PRIMOS responds by raising a condition. For each type of error, a corresponding condition exists.

When a condition is raised, PRIMOS activates the condition-handling mechanism. The condition handler notes what condition exists, then calls an error-handling routine known as an "on-unit" to deal with the error that has occurred.

PRIMOS supplies a default on-unit for each condition. A programmer can specify his own response to a condition by supplying an on-unit of his own. When a condition occurs for which a programmer-supplied on-unit exists, the actions specified in the on-unit will be taken, rather than those specified in the PRIMOS default on-unit.

Information on the system default on-units and the method for substituting programmer-supplied on-units is contained in The Prime User's Guide. For complete information on the condition handler, see The PRIMOS Subroutines Guide.

CONVENTIONS USED IN THIS GUIDE

Various conventions are used in the following sections. Their meanings must be clearly understood by the reader.

Conventions Indicating Extensions

Every F77 extension is labeled as such in the text of this guide.

When a specific feature is explicitly described as being an F77 extension, the implication is that it is not part of FORTRAN 77. No such feature should be used in a program which may have to run on a non-Prime system.

When the F77 language is mentioned in general, the reference is to Prime's extended FORTRAN 77 as a whole.

Conventions in Examples

In all examples involving dialog between the user and the system, the user's input is underlined, and the system's output is not. For example:

```
OK, attach mydirec
OK, ed oldfile
EDIT
```

Examples consisting only of F77 statements, with no responses from the

system, are not underlined.

```
COMMON // A
CALL SUB(A)
STOP
END
```

Typographical Conventions

WORDS-IN-UPPER-CASE	Uppercase letters identify command words or keywords. They are to be entered literally.
words-in-lower-case	Lowercase letters identify options or arguments. The user substitutes an appropriate numerical or text value.
Brackets [ ]	Brackets indicate that the item enclosed is optional.
Braces { }	Braces indicate a choice of options or arguments. Unless the braces are enclosed by brackets, one choice must be selected.
Parentheses ( )	When parentheses appear in a statement format, they must be included literally when the statement is used.
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.

SECTION 2

FORTRAN 77 LANGUAGE ELEMENTS

DEFINITIONS

In the following sections, a few terms occur repeatedly. The reader must be clear on their exact meanings if discussions using them are to be understood correctly. Also be sure to read the definitions at the beginning of Section 1 for an explanation of the naming conventions used in this book for the various versions of FORTRAN.

Term	Definition
Actual Argument:	A data item passed to a subprogram. Actual arguments appear in the argument list of a subroutine CALL statement or a function reference.
Arithmetic Expression:	Any expression which evaluates to type INTEGER, REAL, DOUBLE PRECISION, or COMPLEX.
Character Expression:	A single item of type CHARACTER, or the concatenation of any number of such items. Substrings and references to CHARACTER functions are permitted. Trailing blanks are of no significance in a character expression.
Dummy Argument:	A variable or array name appearing in the header statement or an ENTRY statement of a subprogram. When the subprogram is invoked, each dummy argument is associated with the actual argument whose name appears in the corresponding position in the CALL statement or function reference.
Fixed-Length Character Expression:	A character expression in which no operand is a dummy argument with an adjustable(*) length specification.
Integer Expression:	Any expression which evaluates to type INTEGER, either directly or after type conversion via the functions INTS, INTL, or INT.

Integer Constant Expression:	Any expression consisting only of integer constants and named integer constants with arithmetic operators and parentheses.
Program unit:	A main program, external function, subroutine, or block data unit.
Segment:	A 128K-byte block of address space.
Subprogram:	Any program unit except a main program.

LEGAL CHARACTER SET

Any ASCII character may appear in FORTRAN 77 character data, Hollerith constants, and I/O files. In program source statements, the legal characters are:

- The 26 uppercase letters:
  A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
- The 26 lowercase letters (F77 Extension):
  a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
- The 1Ø digits: Ø,1,2,3,4,5,6,7,8,9
- These 13 special characters:

  = equals
  ' single quote (apostrophe)
  : colon
  + plus
  - minus
  * asterisk
  / slash
  (left parenthesis
) right parenthesis
  , comma
  . decimal point
  $ dollar sign
  _ underscore (F77 extension. Backarrow on some terminals.)

- Blanks or spaces

Blanks in character and Hollerith constants and in $INSERT statements are treated as character positions. Elsewhere in FORTRAN 77 source text, blanks have no meaning and can be used as desired to improve program legibility. Lowercase letters are mapped to uppercase (except within Hollerith and CHARACTER constants) unless the program is compiled with the -LCASE option. Keywords must be in uppercase if -LCASE is given. The ASCII collating sequence is used. (See Appendix D.)

LINE FORMAT

Each program line is a string of 1 to 72 characters. Each character position in the line is called a column. Columns are numbered from left to right starting with 1. There are three types of lines: Comments; FORTRAN 77 statements (and their continuations); and Insert statements.

In all line types, columns 73-8Ø are available for line order sequence numbers or other identification. (Usage is optional.) These columns, like comments, are ignored by the compiler, but are printed in the program listing.

Comments

Comment lines are identified by the letter "C" or an asterisk in column 1. The remainder of the line may contain anything. A comment line is ignored by the compiler, except that it is printed in the source listing. In F77, a comment may be placed anywhere after Column 6 in a statement line, except inside a character constant, using the format:

```
/* comment */
```

The end of the line terminates the comment and makes the */ unnecessary. A line blank through column 72 is a comment line.

Statements

In the first line of a statement, columns 1-5 are reserved for the statement label, if any. Blanks and leading zeros are ignored. Column 6 must be a blank or a zero. Columns 7-72 contain the statement. The statement may begin with leading blanks, to make the program easier to read. In the continuation of a statement, columns 1-5 must be blank, column 6 may be any character except Ø or a blank, and the statement continuation is in columns 7-72. There may be at most 19 continuation lines.

Inserts

F77 allows files to be inserted automatically into the source file at compile time, via the Insert statement. An Insert statement consists of the keyword $INSERT beginning in Column 1, followed by the pathname of the file to be inserted. See INSERT Statement in Section 3 for more information.

DATA TYPES

Six major data types exist in FORTRAN 77: Integer, Real, Double Precision, Complex, Logical, and Character. Each of these may exist in any of four forms: Constant, Parameter, Variable, or Array. In addition, there are statement labels and Hollerith constants. Some subtypes exist, differing from each other only in storage size, as shown below.

Type	Bytes	Range
INTEGER	2 or 4	Same as for INTEGER*2 or INTEGER*4. (See below.)
INTEGER*2 (short integer)	2	-(2**15) to (2**15-1) Decimal -32768 to 32767 Octal :Ø to :177777
INTEGER*4 (long integer)	4	-(2**31) to (2**31-1) Decimal -2147483648 to 2147483647 Octal :Ø to :377777777777
REAL (REAL*4)	4	± (1Ø**-38 to 1Ø**38)
DOUBLE PRECISION (REAL*8)	8	± (1Ø**-9ØØ2 to 1Ø**9825)
COMPLEX (COMPLEX*8)	4+4	Each component has same range as REAL
COMPLEX*16	8+8	Each component has same range as DOUBLE PRECISION
LOGICAL	2 or 4	T or F
LOGICAL*4	4	T or F
LOGICAL*2	2	T or F
LOGICAL*1	1	T or F
CHARACTER	1 to 32767	1 to 32767 characters.
Statement Label	2 or 4	1 to 99999
Hollerith	Varies	1 to 256 characters

The types INTEGER*2, LOGICAL*2, LOGICAL*1, and COMPLEX*16 are F77 extensions. The names INTEGER*4, REAL*4, REAL*8, COMPLEX*8, and LOGICAL*4 are F77 synonyms for the corresponding FORTRAN 77 data types, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL. These synonyms, and the types INTEGER*2, LOGICAL*2, and LOGICAL*1, are provided for upward compatibility of existing FORTRAN IV programs: they should not be used in new programs.

With the exception of the CHARACTER data type, a new feature of FORTRAN 77, the FORTRAN 66 and FORTRAN 77 data types are the same. Since the reader is expected to be familiar with some version of FORTRAN IV (extended FORTRAN 66), only highlights and Prime extensions of the FORTRAN 77 data types are mentioned below. The CHARACTER type is discussed in more detail. Each data type is illustrated with several constants of that type.

INTEGER Data

An INTEGER data item represents an integer exactly. Integers are always written without a decimal point. An integer constant may be represented in decimal or octal form. (Octal form is an F77 extension.)

Decimal	Octal	
-2Ø4	-:314	(same as :37777777464)
Ø	:Ø	
8	:1Ø	
1911	:3567	

F77 supports two integer subtypes: INTEGER*2 (short) and INTEGER*4 (long). When a variable is declared of type INTEGER with no *(size) specified, or becomes type INTEGER by default, the variable will either be INTEGER*4 if the program is compiled with -INTL (the default), or INTEGER*2 if it is compiled with -INTS.

Integer constants compiled under -INTL also become INTEGER*4. Under -INTS, they become INTEGER*2 unless:

- Their magnitude lies outside the range ±32767 or is greater than :177777.
- Their representation, including leading zeroes, contains more than 5 decimal or 6 octal digits. Example:

3Ø	short integer constant (under -INTS)
ØØØØ3Ø	long integer constant (always)

Within a program, long and short integers are interchangeable. They may be mixed freely in expressions, though care must be taken that short integers are not assigned values outside their range. When a program communicates with pre-existing library and I/O routines, integer arguments supplied to those routines must be of the type they expect.

Some library routines require INTEGER*2 arguments. In these cases, convert any long-integer arguments to short integer via the INTS function (not to be confused with the -INTS compiler option, which affects all integer data) or if appropriate compile the whole program with -INTS. See the PRIMOS Subroutines Reference Guide, PDR3621, for information on library subroutines.

REAL Data

A REAL data item is an approximation to a real number. REAL data is always written with a decimal point, an exponent, or both. The decimal point is optional if an exponent is given. Blanks may appear between the mantissa and its exponent.

-204. -20400 E-2 0. 8.8756E4 8.8756E+4

Up to seven significant digits are retained. Exponents may range from -38 to +38.

Real constants must fall in the type REAL range. They will not become DOUBLE PRECISION on the basis of magnitude or number of digits.

DOUBLE PRECISION Data

DOUBLE PRECISION data is also called REAL*8. It is similar to REAL except that twice as much storage is allocated, and "D" rather than "E" appears in the exponent. The "D" exponent is mandatory. Examples:

123456789.D0 2.5 D-2 0.D0 -999D+21

Up to 14 significant digits are retained. The exponent may range from -9902 to +9825.

COMPLEX Data

A COMPLEX (or COMPLEX*8) data item is an ordered pair of real numbers. The first number represents the real part, the second the imaginary part. In a complex constant, or when a complex number is used in list-directed I/O, the number appears in parentheses with its components separated by a comma. Examples:

(1.,1.) (25E6, 331.) (.172E19, 304E-2)

The comma and parentheses must appear when a complex number is used in list-directed I/O. They must be omitted from a complex number used in formatted I/O.

COMPLEX*16 Data

The COMPLEX*16 data type is identical to COMPLEX except that DOUBLE PRECISION numbers are used rather than REAL numbers.

LOGICAL Data

LOGICAL data items denote only the logical values TRUE and FALSE. In programs, logical constants must be written:

.TRUE. .FALSE.

In input files, either the constants or the letters T and F may denote the values. On output, T and F are always written.

Logical constants and logical variables lacking a *(size) specification become either LOGICAL*4 if the program is compiled with -LOGL (the default), or LOGICAL*2 if it is compiled with -LOGS. A LOGICAL*1 type is also provided for compatibility with IBM FORTRAN. This type should not be used in new programs, because it is processed less quickly than LOGICAL*2 or LOGICAL*4.

CHARACTER Data

The CHARACTER data type is a new feature of FORTRAN 77. It makes Hollerith strings and the use of arithmetic variables to hold character data obsolete. F77 continues to support the Hollerith and arithmetic/character techniques as an aid to upward compatibility of existing programs. New programs should use only CHARACTER data.

A CHARACTER data item is a nonempty string of characters. Each item has a length equal to the number of characters it contains. The character positions are numbered from 1 to LENGTH. Each character occupies one byte.

A character constant consists of a string of characters enclosed in single quotes. Any internal single quotes must be represented by two consecutive single quotes. The two count as only one character position.

'THAT''S ALL' occupies ten positions.

Declaration: The form of a CHARACTER type-statement is:

CHARACTER [*len] cname [,cname]...

where len is an integer constant expression giving the length of the CHARACTER variable. If *len is omitted, the length defaults to 1. In a dummy argument in a subprogram, len may be replaced by an asterisk in parentheses. A character item so declared will take on the length of the corresponding actual argument in the invoking program unit.

CHARACTER entities may be initialized in a type-statement. CHARACTER entities having different lengths may be declared in the same type-statement. See Type-Statements in Section 3 for details.

CHARACTER parameters and arrays are declared as with other data types.

Substrings: A contiguous subset of a CHARACTER data item is known as a substring. A substring of a variable or array element is specified:

```
VARNAME(L:H) or ARRAYNAME(subscripts)(L:H)
```

where L and H are integer expressions giving the lowest and highest character positions of the desired substring. If 1 is omitted, 1 is assumed. If H is omitted, the length of the variable is assumed.

Substrings cannot be extracted from constants and parameters. When a substring of a constant or parameter is needed, assign the constant or parameter to a CHARACTER variable, then extract the substring from the variable.

Suppose CVAR = 'ABCDE'. Then:

```
CVAR (2:5) = 'BCDE'
CVAR (:3) = 'ABC'
CVAR (4:) = 'DE'
```

Concatenation: Character entities may be linked together using the concatenation operator, written '//'.

```
'ABC' // 'XYZ' = 'ABCXYZ'
'Z' // CVAR (2:5) = 'ZBCDE'
```

Assignment: Character entities may be assigned using the "=" sign. Where lengths do not match, truncation or padding with blanks takes place on the right. Undefined positions on either side of positions assigned by substring remain undefined.

In FORTRAN 77, no position may act as both source and destination in a substring assignment. F77 relaxes this restriction. This extension must be used carefully, because the source string is not copied before execution of a substring assignment. The assignment may therefore encounter its own effects partway through execution.

If K and Q are CHARACTER*5:

```
K = 'A' // 'B' // 'C'          /* K = 'ABCbb'
Q (3:4) = K (2:3)              /* Q = '??BC?'
K = K // K                     /* K = 'ABCbb'
K (1:3) = K (2:4)              /* K = 'BCbbb'
```

Comparison: Character entities may be compared using the relational operators. The ASCII collating sequence is used. (See Appendix D.)

```
IF ('ABX' .LT. (CVAR (2:3)//'ZQ')) GO TO 1ØØ
```

Intrinsic Functions: Various intrinsic functions exist to provide services related to CHARACTER data items. They are described in Section 6.

Input/Output: I/O of CHARACTER data is similar to I/O for the other data types. Formatted CHARACTER I/O uses the "A" field descriptor. See Section 4.

Statement Labels

Statement labels exist in three forms:

- Any statement may have a label between 1 and 99999 affixed to it, in columns 1-5 of its first line.
- An integer variable becomes a statement label variable when the integer variable is set to a label value by an ASSIGN statement. The variable may then be used in an assigned GO TO. See the ASSIGN and ASSIGNED GO TO statements in Section 3.
- A statement label prefixed by the character "*" or "$" becomes a statement label constant. (Use of "$" for this purpose is an F77 extension, and is considered an obsolete technique.) One or more such constants may be placed in the argument list to a subroutine, permitting the subroutine to return to one of the lines whose label it has received, rather than to the line following the subroutine call. See Section 5.

Hollerith Constants

Hollerith constants are accepted in F77 to aid upward compatibility of FORTRAN IV programs. This type is obsolete. Use CHARACTER constants and variables when writing new programs.

OPERANDS

Operands are those elements which are manipulated by the program. Four types of operand exist in FORTRAN 77: Constants, Parameters, Variables, and Arrays.

Constants

Constants exist for every data type. In a program, a constant appears as a literal representation of the desired value. The compiler determines the type of the constant from its appearance, its context, and the compiler options in effect. Appropriate storage is allocated, and the value is stored in it.

The correct form for each type of constant appears in the previous

subsection under the appropriate data type.

Parameters

Parameters are named constants, and may be of any data type. They are functionally similar to constants, but are referenced by the name assigned to the value in a PARAMETER statement, rather than by a literal occurrence of the value. Parameters may not appear in FORMAT statements. Parameter names follow the same rules as variable names.

Do not confuse parameters with arguments to subroutines. In FORTRAN 77 the term "parameter" denotes only a named constant.

Variables

Variables are data items whose values may be assigned, and subsequently altered, during program execution.

FORTRAN 77 variable names contain from 1 to 6 characters. In F77, variable names may have from 1 to 32 characters. Character 1 must be alphabetic; characters 2-32 (if any) must be alphanumeric, or the characters "$" or "_". Users are discouraged from using "$" in their variable names because this character is used extensively in Prime-supplied software names, where it serves to implement a system of naming conventions.

When no type is explicitly declared, a variable whose name begins with the letters I through N becomes type INTEGER, and a variable whose name begins with A-H or O-Z becomes type REAL. See Section 3 for instructions on how to override this implicit convention, and how to specify DOUBLE PRECISION, COMPLEX, CHARACTER, and LOGICAL types.

Arrays

Arrays are ordered, multidimensional sets of variables. An array is declared in a DIMENSION, COMMON, or type-statement such as:

```
DIMENSION array declarator [,array declarator]...
```

where each "array declarator" has the form;

```
ANAME (d1[,d2]...[,d7])
```

in which ANAME is the name of an array (same rules as for a variable name), and each dn has the form:

```
[Ln:]Hn
```

Ln is the lower subscript bound, and Hn is the upper subscript bound, for dimension n. There may be at most seven dimensions. If Ln is omitted, it is assumed to be 1.

Example:

```
INTEGER ARR(-3:3,7,Ø:2Ø4,-2Ø7:-91,81)
```

In a main program, Ln and Hn must be integer-constant expressions. For a dummy argument array in a subprogram, they may be integer expressions (for an adjustable array), and the upper bound of the last dimension may be given as an asterisk (for an assumed-size array). See Section 5 for details. Arrays are stored by columns: the leftmost subscript varies most rapidly when the array is accessed in storage order.

Referencing Arrays

Array references have the form:

```
ANAME (S1[,S2]...[,S7])
```

where each Sn is a subscript expression.

A subscript expression is any legal FORTRAN 77 integer-valued expression. It may contain constants, variables, function references, intrinsic references, and other array references.

Note

Non-integer data items are not allowed in subscript expressions. Convert any such items to integers using the appropriate conversion function (IDINT, IFIX, INT, etc.)

An array longer than one segment (128K bytes) must be stored in a COMMON block. An array shorter than one segment should not be stored in a COMMON block longer than one segment. See ARRAYS AS ARGUMENTS in Section 5 for more information. See the COMMON Statement in Section 3 for a restriction on the placement of data items (including arrays) in a COMMON block.

Evaluation of a function reference in a subscript expression must not alter any other elements of the subscript expression list, either directly or by altering arguments used in other function references.

Caution

When an array that crosses or may cross a segment boundary is passed as an argument to a subprogram, special action is necessary. See ARRAYS AS ARGUMENTS in Section 5.

OPERATORS

Operators modify an operand, or combine or concatenate two operands.

Logical Operators

FORTRAN 77's logical operators are: .NOT., .AND., .OR., .EQV., and .NEQV.. In the following, P and Q are of type LOGICAL.

.NOT.: (.NOT.Q) negates the value of Q.

Q	.NOT. Q
.TRUE.	.FALSE.
.FALSE.	.TRUE.

.AND.: (P .AND. Q) is the logical ANDing of P and Q (set intersection)

	P	
Q	.TRUE.	.FALSE
.TRUE.	.TRUE.	.FALSE.
.FALSE.	.FALSE.	.FALSE.

.OR.: (P .OR. Q) is the logical non-exclusive ORing of P and Q. (Set union)

	P	
Q	.TRUE.	.FALSE.
.TRUE.	.TRUE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.

.EQV.: (P .EQV. Q) is the logical equivalence of P and Q.

	P	
Q	.TRUE.	.FALSE.
.TRUE.	.TRUE.	.FALSE.
.FALSE.	.FALSE.	.TRUE.

.NEQV.: (P .NEQV. Q) is the same in effect as (.NOT. (P .EQV. Q)). It acts as an exclusive or.

	P	
Q	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.

Arithmetic Operators

**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction or Unary Minus
=	Assignment

Relational Operators

.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Character Operator

//	Concatenation

Operator Priority

**	Exponentiation
-	Unary Minus
* or /	Multiplication or division
+ or -	Addition or subtraction
//	Concatenation
.LT. .LE. .EQ. .NE. .GT. .GE.	Relational operators (All have same priority.)
.NOT.	Logical negation
.AND.	Logical intersection
.OR.	Logical union
.EQV. .NEQV.	Logical equivalence/nonequivalence

TYPE CONVERSION

Logical operators may combine logical operands of differing storage lengths, and arithmetic operators may combine operands of differing numeric types. The type of the result in such cases depends on the types of the operands.

Logical Conversion

The storage length of the result when logical data of differing lengths are combined is the longer of the two lengths. Example:

(LOGICAL*2 .AND. LOGICAL*4) is LOGICAL*4

Arithmetic Conversion

The type of the result when differing numeric types are combined will be that of the operand having the higher type in the following list:

COMPLEX*16
COMPLEX*8
DOUBLE PRECISION
REAL
LONG INTEGER
SHORT INTEGER

For Example REAL + SHORT INTEGER is a REAL

Special Case: To prevent loss of precision, the result-type when COMPLEX*8 and DOUBLE PRECISION data are combined will be COMPLEX*16. (F77 extension).

Caution

When long integers are converted to reals, there may be a loss of precision. No error message will be generated, but incorrect results may occur.

ORDER OF EVALUATION

When operators having the same priority appear successively in an expression, the value of the expression may depend on the order in which the operators are processed. The order is sometimes different in different programming languages, and in different compilers for the same programming language. This variability is a common cause of programming errors.

In F77, multiple exponentiations are processed from right to left: A**B**C = A**(B**C). For the other operators, multiple occurrences of operators of equal priority are generally processed left to right: A*B/C = (A*B)/C. However, the compiler takes advantage of groupings of elements (in accordance with mathematical rules) to optimize its output. For example, given A*B - A*C the compiler may evaluate A*(B-C) instead. Consequently, evaluation may sometimes not be strictly left to right.

The compiler always respects the integrity of parentheses. For example, (A*B) - (A*C) would be evaluated exactly as written. Expressions within parentheses are always evaluated before expressions outside them. For example, A*(B/C) will have its quotient evaluated first. Where evaluation order is critical, use parentheses to eliminate any ambiguity.

Where multiple references to functions occur in an expression, the compiler may evaluate them in any order. No function reference may alter any other value in the expression, either directly or by altering arguments used in other function references.

PROGRAM COMPOSITION

Each program unit consists of a number of program lines. Program lines are grouped and ordered as shown in Table 2-1. Vertical boundaries in the table denote classes of statements that can be interspersed. Horizontal boundaries denote classes of statements that cannot be interspersed. F77 statements are discussed in Section 3 and Section 4.

Any number of program units may be present in a single file. Only comments may appear between the END statement of one program unit and the header statement of the next.

In F77, no block of executable code can cross a segment boundary. Therefore, no program unit may produce more than 128K bytes (one segment) of code. Rarely if ever will a program unit be any larger than this; one that is must be broken up. Program data is kept in separate data segments, and hence does not compete for space with the executable code.

The names of F77 program units may not be more than 8 characters long. Additional characters will be ignored and a warning message printed.

Table 2-1. FORTRAN 77 Program Composition

<table>
<tr><td rowspan="5">COMMENT LINES</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, OR BLOCK DATA STATEMENT</td></tr>
<tr><td rowspan="4">FORMAT AND ENTRY STATEMENTS</td><td rowspan="2">PARAMETER STATEMENTS</td><td>IMPLICIT STATEMENTS</td></tr>
<tr><td>OTHER DATA DEFINITION STATEMENTS</td></tr>
<tr><td rowspan="2">DATA STATEMENTS</td><td>STATEMENT FUNCTION STATEMENTS</td></tr>
<tr><td>EXECUTABLE STATEMENTS</td></tr>
<tr><td colspan="4">END STATEMENT</td></tr>
</table>

SECTION 3

PROGRAM SPECIFICATION STATEMENTS

All FORTRAN 77 statements concerned with specifying a FORTRAN program, as distinct from specifying its I/O interface, are discussed in this section. Because of the complexity of FORTRAN I/O and the large number of new features added in FORTRAN 77, I/O statements are dealt with separately in Section 4.

SUMMARY OF STATEMENTS

FORTRAN program specification statements are listed below with their functional categories. All *(length) suffixes to statements other than the CHARACTER data definition statement, and all Compiler Control statements, are F77 extensions.

Header Statements

PROGRAM
SUBROUTINE
FUNCTION
ENTRY (Secondary header)
BLOCK DATA

Data Definition Statements

IMPLICIT
INTEGER [{*2,*4}]
REAL [{*4,*8}]
DOUBLE PRECISION
COMPLEX [{*8,*16}]
LOGICAL [{*1,*2,*4}]
CHARACTER [*n]
DIMENSION
PARAMETER

Data Initialization Statement

DATA

Storage Allocation Statements

COMMON
EQUIVALENCE
SAVE

Procedure Statements

CALL
EXTERNAL
INTRINSIC
Statement Function

Compiler Control Statements

NO LIST
LIST
$INSERT

Assignment Statements

Arithmetic assignment
Logical assignment
Character assignment

Control Statements

DO
CONTINUE
ASSIGN
unconditional GO TO
computed GO TO
assigned GO TO
arithmetic-IF
logical-IF
block-IF
PAUSE
RETURN
STOP
END

HEADER STATEMENTS

Header statements define primary and secondary entry points to program units, and perform other functions as indicated. The name in a header statement must be constructed according to the rules for variable names. See Section 2.

The names of F77 program units must not be more than 8 characters long. Additional characters will be ignored, and a warning message printed.

PROGRAM Statement

```
PROGRAM name
```

The PROGRAM statement gives a name to a main program. It is not required. If present, it must be the first statement of the main program.

The name must not duplicate the name of any common block or subprogram, or of any data item in the main program.

When a PROGRAM name has been specified, that name will appear in any SEG load map in place of "####", and will be used by the Symbolic Debugger in place of "$MAIN".

SUBROUTINE Statement

```
SUBROUTINE name [( [argument [,argument]...] )]
```

Declares a program unit to be a subroutine, assigns its name, and specifies its dummy arguments. See Section 5.

FUNCTION Statement

```
[type] FUNCTION name ( [argument [,argument]...] )
```

Declares a program unit to be a function, assigns its name and type, and specifies its dummy arguments. If no type is declared in the FUNCTION statement, the typing can be done in an ordinary type-statement. If no type is declared anywhere, default typing will occur. An IMPLICIT statement in a function affects default typing of the function name. See Section 5 for more information.

ENTRY Statement

```
ENTRY entry-name [( [argument [,argument]...] )]
```

Specifies a secondary entry point in a subprogram, assigns its name, and specifies its dummy arguments. See Section 5.

BLOCK DATA Statement

```
BLOCK DATA [name]
 .
 .
 .     statements
 .
 .
END
```

The BLOCK DATA statement designates and optionally names a BLOCK DATA subprogram. A program may contain any number of such subprograms.

A BLOCK DATA subprogram initializes data items in named or blank COMMON blocks. Initialization of blank COMMON is an F77 extension. The entire block must be specified in a COMMON statement in the subprogram if any part of it is to be initialized. Only COMMON, EQUIVALENCE, DIMENSION, DATA, IMPLICIT, PARAMETER and type-statements may appear.

DATA DEFINITION STATEMENTS

These statements create and control the properties of the variables, arrays, and parameters which constitute the data elements of a program.

FORTRAN 77 automatically assigns types to all variables, parameters, arrays, and functions that do not appear in type-statements. The FORTRAN 77 language default is as follows: if the symbol's first character is I through N (inclusive), the symbol is typed as integer; all others (A-H, O-Z) are typed as real. The default integers are long integers unless the program is compiled with the short integer option -INTS. See Section 2 and Section 7. The type of an intrinsic function is predefined, and does not depend on this implicit typing mechanism.

IMPLICIT Statement

```
IMPLICIT type (list) [,type (list)]...
```

The IMPLICIT statement allows the programmer to override the language convention for default data-typing by first letter. Each type is a data type such as REAL*4, COMPLEX, etc. Each list lists the letters which will cause default to that type. Letters may be separated by a comma, or an inclusive group of letters may be indicated with a dash.

Symbols not typed in a type-statement or by a default specified in an IMPLICIT statement will be typed by the FORTRAN 77 language default. Example:

```
IMPLICIT DOUBLE PRECISION (A,N,O,P-Z), LOGICAL (B), CHARACTER*3 (M)
```

First letter of symbol	Type
A, or N through Z	Double Precision
B	Logical
C through H	Real
I through L	Integer
M	Character*3

If used, the IMPLICIT statement must be the first statement of a main program (second if a PROGRAM statement exists), or the second statement of a subprogram. IMPLICIT affects all symbols not otherwise typed. This includes dummy arguments in the header statement of a subroutine or function, and function names which are not explicitly typed. The

user should take care to make sure that these implicitly typed arguments will be of the proper data type. IMPLICIT typing does not affect intrinsic functions.

Type-Statements

```
type v [,v]...
```

The type-statement allows override of the implicit type assignments of symbol names which would be done either by IMPLICIT or by language default. A data item may be initialized in a type-statement.

The word type is replaced by one of the data-type specifications:

```
INTEGER
INTEGER*2
INTEGER*4

REAL
REAL*4 (same as REAL)
REAL*8 (same as DOUBLE PRECISION)

DOUBLE PRECISION (same as REAL*8)

COMPLEX
COMPLEX*8 (same as COMPLEX)
COMPLEX*16

LOGICAL
LOGICAL*1
LOGICAL*2
LOGICAL*4

CHARACTER (same as CHARACTER*1)
CHARACTER*n
```

Where n (in CHARACTER*n) is an integer constant, an integer constant expression in parentheses, or an asterisk in parentheses (for adjustable dummy argument length in a subprogram), and the v's are a list of variable names, parameter names, array names, function names, or array declarators.

The types INTEGER*2, COMPLEX*16, LOGICAL*1, and LOGICAL*2 are F77 extensions. The names INTEGER*4, REAL*4, REAL*8, COMPLEX*8, and LOGICAL*4 are F77 synonyms for the corresponding FORTRAN 77 data types INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL. These synonyms are provided for upward compatibility of existing FORTRAN IV programs: they should not be used in new programs.

The storage length given in the type will ordinarily apply to all the data items in the statement. In F77, lengths may also be specified for data items singly. When both a single and a general length specification are given, the single specification takes precedence.

For example:

```
INTEGER  A*4, B*2
INTEGER*4  C, D*2, E
CHARACTER*50  F, G*100, H, I, J*1
```

is equivalent to:

```
INTEGER*4  A,C,E
INTEGER*2  B,D
CHARACTER*1  J
CHARACTER*50  F,H,I
CHARACTER*100  G
```

Recognition of synonymous data types is provided to ease conversion of existing programs to F77. INTEGER will normally default to INTEGER*4 (long integer) unless the program is compiled with the -INTS option, in which case it will default to INTEGER*2 (short integer). LOGICAL will default to LOGICAL*4 unless the program is compiled with -LOGS, in which case it will default to LOGICAL*2. See Section 7 for compiler option information.

To initialize a data item in a type-statement, enclose the desired value between slashes and insert it immediately after the data item name. The rules and syntax of the DATA statement apply, except that each initializing value must follow its data item immediately, and not all the items need be initialized. Example:

```
INTEGER A/5/,B,C,D,E(2)/1,2/,F(5)/5*10/
```

Any initialized data item will be placed in static storage.

DIMENSION Statement

```
DIMENSION array declarator [,array declarator]...
```

Where each array declarator is as described under ARRAYS in Section 2.

A DIMENSION statement declares a symbolic name typed in a type-statement, or by default, to be an array, and sets the number of dimensions and bounds of each dimension of the array.

A list of arrays can be declared and typed in one statement by replacing the keyword DIMENSION above with any data-type specifier.

PARAMETER Statement

```
PARAMETER (p=c [,p=c]...)
```

The p's are symbolic names previously typed in any standard way, or by default. Each c is a constant expression of a type appropriate to the corresponding p. A constant expression consists only of constants, parameters, constant expressions in parentheses, and appropriate operators. Any parameters that appear must have been defined in a previous PARAMETER statement. Function references and non-integral exponentiations are prohibited. A parameter may not be used to form a complex constant.

Unless specifically prohibited, parameter names may be used wherever a constant could be used (including DATA and DIMENSION statements) except in FORMAT statements. Since parameters are named constants, they may not be elements of COMMON blocks and cannot be equivalenced. They may be used in declaring bounds of arrays in COMMON.

In F77 the parentheses around the parameter list may be omitted.

DATA INITIALIZATION STATEMENT

DATA Statement

```
DATA k/d/ [,k/d/]...          (Commas are optional)
```

Allows initialization of data items at load time. Each k is a list of variables, array names, array elements, substring names, and implied-DO lists, in which any expressions that appear must be integer constant expressions. Each d is a list of constants and parameters, possibly with repetition factors. A repetition factor is an integer constant followed by an asterisk.

The values in each d are assigned in order to the corresponding items in k. For each item, there must be a value of a type legally assignable to the item. Numeric type conversion and character padding/truncation will occur as they would in an assignment statement. Any implied-DO lists and repetition factors present operate as they would in a list-directed READ statement. Example:

```
INTEGER A,K, ARR(1:5, 1:5)
DATA A,K/3,4/((ARR(I,J), I=1,5), J=1,5)/25*5.Ø/
```

When large arrays of character data must be initialized, effort can be saved by declaring a separate CHARACTER variable equal in length to the entire array, equivalencing it to the array, and initializing it with the concatenation of all the desired initial values. Example:

```
CHARACTER*2 K(3)
CHARACTER*6 INITK
EQUIVALENCE (K,INITK)
DATA INITK /'ABCDE'/
/* K(1)='AB'   K(2)='CD'   K(3)='E ' */
```

Any data item initialized in a DATA statement, and any data items equivalenced to it, will be declared static by the compiler. See the SAVE statement above and the -SAVE option in Section 7.

F77 also allows data to be initialized in type-statements. See above.

STORAGE ALLOCATION STATEMENTS

F77 provides two forms of storage: static and dynamic. Static storage is allocated for all program units before program execution begins, while dynamic storage is allocated only when a subprogram becomes active, and is de-allocated when it becomes inactive. Consequently, static data items retain their values between subprogram references, while dynamic data items lose their values when a subprogram returns. Dynamic data items in a main program never lose their values, because the main program is always active.

The following factors determine whether a given data item will be static or dynamic:

- If a program unit is compiled with the -SAVE option, all data items in it will be static. If it is compiled with -DYNM (the default), all data items not otherwise declared static will be dynamic.

- All data items in any COMMON block are static in F77. (In FORTRAN 77, only blank COMMON is static.)

- Any data item may be made static by naming it in a SAVE statement.

- Any data item initialized in a DATA statement will be declared static, since only static storage can retain an initial value.

To save space in memory, no data item should be declared static unless there is a specific reason to do so.

The following constraints apply to F77 storage allocation.

- A program unit cannot have more than 128K bytes (one segment) of local static storage. If more is needed, put the data items in a COMMON block.

- A program unit cannot have more than 128K bytes of dynamic storage. Since all COMMON blocks are static, this figure cannot be increased: when more than 128K bytes of program data exist, the excess will have to be made static.

- Due to the above, any array longer than 128K bytes must be kept in a COMMON block.

COMMON Statement

COMMON /x/a [,/x/a]... (Commas are optional)

Where each a is a non-empty list of variable names or array names separated by commas, and each x is a COMMON block name or is empty (blank COMMON). A COMMON block name must not be identical with the name of any user-supplied or F77 library subprogram. A COMMON block name must not contain more than eight characters.

The same name, or no name, may appear more than once in a COMMON statement, and in more than one COMMON statement. Data items are assigned sequentially within a COMMON block in the order of appearance in the COMMON statement(s) defining the block. SEG assigns all COMMON blocks with the same name to the same storage area, regardless of the program or subprogram in which they are defined.

The length of a COMMON block is the number of bytes used by all the items specified in the COMMON statement(s), plus the number of bytes appended to the block by any EQUIVALENCE statements. COMMON blocks with the same block name (or no name) in different program units are not required to have elements within the blocks agree in name, type, or order.

Blank COMMON blocks may be of differing lengths. In FORTRAN 77, all instances of a named COMMON block must have the same length. This restriction is relaxed in F77, as an aid to compatibility with other extended versions of FORTRAN 77.

When a given COMMON block, named or blank, has different lengths in different program units, the program unit containing the longest instance of the block must always be loaded first, because SEG allocates space for a COMMON block on the basis of its first occurrence. Note that a set of program units with COMMON blocks could easily be generated for which no correct load order exists. The preferred method is simply to make all instances of a COMMON block the same length, by padding them as necessary. No inefficiency of time or space utilization can result from following this practice.

Two restrictions exist on the layout of data items in a COMMON block.

- In any COMMON block, all data items except CHARACTER and LOGICAL*1 variables and array elements must begin at a word boundary (Ø, 2, 4... bytes from storage location Ø). Use padding variables as needed to maintain word alignment.

- In large COMMON blocks - those over 128K bytes (one segment) long - a segment boundary will fall somewhere in the COMMON block. No data item, including a COMPLEX item, may be split between two segments. An array may span a segment boundary so-

long as the boundary falls between array elements. (See the Note below.)

To insure that no data item in a large COMMON block will be split at a segment boundary, the F77 compiler enforces the following restrictions:

Every CHARACTER array more than one segment (128K bytes/characters) long, which length requires that it be kept in a COMMON block, must have an element length that is a power of two.

Every variable and array of any kind in a large COMMON block must be offset by a multiple of its element length from the start of the COMMON block.

Note

When a COMMON block over one segment long contains an array, and that array is passed to a subprogram as an actual argument, the subprogram must have been compiled with -BIG. An array less than 128K bytes long should not be placed in a COMMON block more than 128K bytes long. See ARRAYS AS ARGUMENTS in Section 5.

EQUIVALENCE Statement

```
EQUIVALENCE (k ,k [,k]... ) [,(k ,k [,k]... )]...
```

Where each k is a variable, array element, or array name. When an unsubscripted array name is mentioned, the effect is as if its first element had been mentioned. Subscripts must be integer constant expressions. FORTRAN 77 requires a separate subscript for each dimension of an array. F77 allows one subscript to be used for the whole array, indexing it in storage order, as in FTN.

An EQUIVALENCE statement causes all the items mentioned in each parenthesized list to be stored beginning with the same byte of physical storage. When variables of different lengths are equivalenced the shorter is stored in the first bytes of the longer. When specific array elements are equivalenced, the arrays as wholes become correspondingly aligned.

When data in a COMMON block is equivalenced to other data, some bytes of the other data may become aligned with storage positions outside of the COMMON block. When this occurs, the block has been extended. Only extensions to the right (towards higher storage addresses) are legal.

Legal example:

```
INTEGER  I, A(3)
COMMON  // I
EQUIVALENCE  (I, A(1))
```

Here A(2) and A(3) extend the common block to the right (towards higher

storage addresses). This is permissible.

Illegal example:

```
INTEGER  I, A(3)
COMMON  // I
EQUIVALENCE  (I, A(3))
```

Here A(1) and A(2) extend the COMMON block to the left. This is illegal; an error message will result.

Data items already fixed in storage cannot be equivalenced. An equivalence statement cannot make self-contradictory demands. Hence the following are all illegal.

```
INTEGER  A(5)
EQUIVALENCE  (A(1), A(5))

COMMON  // A,B
EQUIVALENCE  (A, B)

INTEGER  A(5), B(5), C(5)
EQUIVALENCE  (A(5), B(1)), (B(5), C(1)), (C(5), A(1))
```

Prime's hardware requires that all COMMON block data items except CHARACTER and LOGICAL*1 variables and array elements must begin at a word boundary (Ø, 2, 4...bytes from the start of the COMMON block). No EQUIVALENCE can violate this rule. Hence the following is illegal:

```
CHARACTER*1  CVAR(4)
INTEGER*4  NUM
COMMON // CVAR
EQUIVALENCE  (CVAR(2),NUM)
```

Any data item equivalenced to a static data item will itself be static. In F77, character and non-character data may be equivalenced. FORTRAN 77 does not allow this practice.

SAVE Statement

```
SAVE [v [,v]...]
```

Where each v is a variable or array name that is not part of or equivalenced to a common block. If no v's appear, the SAVE is taken to include all local data items.

The SAVE statement causes the subprogram variables and arrays named in it to retain their values between invocations (static storage) rather than losing their values when the subprogram returns (dynamic storage).

In F77, all COMMON blocks, named or blank, are static in all cases; hence the appearance of a COMMON block name in a SAVE statement has no effect. If a program is compiled with the -SAVE compiler option, all local data items will be static; hence no SAVE statement will have any effect. If a program is compiled with -DYNM, (the default) all local data items will be dynamic unless they are saved.

PROCEDURE STATEMENTS

CALL Statement

```
CALL name [( [argument [,argument]...] )]
```

Where name is a subroutine name and the arguments are a list (possibly empty) of the arguments passed. The CALL statement transfers control to the named subroutine. See Section 5.

EXTERNAL Statement

```
EXTERNAL name [,name]...
```

Where each name is the name of a user-supplied or library subprogram, or is a dummy subprogram name. An EXTERNAL statement allows the subprograms specified to be passed as arguments to other subprograms, where they may be used directly, or declared EXTERNAL and passed again. Without the EXTERNAL statement, variables would be default-declared and passed instead.

Should a name specified as EXTERNAL in a program unit be that of an intrinsic function, the name will refer to the user-supplied subprogram, and the intrinsic will be unavailable to that program unit.

It is recommended that the names of any user-supplied subprograms called from a program unit appear in an EXTERNAL statement in that unit. This method enhances portability to other systems, where some intrinsic function might have the same name as a user-supplied subprogram.

INTRINSIC Statement

```
INTRINSIC name [,name]...
```

Where each name is the name of an F77 intrinsic (built-in) function.

An INTRINSIC statement allows the function names specified to be passed as arguments to other subprograms, which may then reference the particular function passed. Without the INTRINSIC statement, variables would be default-declared and passed instead. No name may appear in both an INTRINSIC and an EXTERNAL statement, or in more than one INTRINSIC statement, in the same program unit.

It is recommended that the names of all intrinsic functions referenced in a program unit be listed in an INTRINSIC statement in that unit. This practice will result in immediate diagnostic messages if the program is run on a different system which does not supply all the needed intrinsics.

Statement Function

Any function which can be expressed in a single assignment statement can be written as a Statement Function. These are discussed in Section 5. A statement-function name may not be passed as an actual argument.

COMPILER CONTROL STATEMENTS

The following statements are F77 extensions. They provide a means of controlling source-listing generation from within a program, and of directing the compiler to insert files into the source program.

NO LIST Statement

```
NO LIST
```

If a source listing of any kind has been specified in the compiler options, encounter of a NO LIST statement will suppress generation of the listing for source lines following the statement.

If no source listing has been specified, NO LIST has no effect.

LIST Statement

```
LIST
```

The LIST statement reverses the effect of a NO LIST statement: source-listing generation resumes (or begins) following the LIST statement.

A LIST statement will not of itself cause source listing to be generated: an appropriate compiler option must have been given. If one was not, LIST has no effect.

FULL LIST Statement

```
FULL LIST
```

This statement is an obsolete equivalent to LIST. It is supported for compatibility with FTN, and should not be used in new programs. See Appendix A for a discussion of FTN/F77 compatibility.

$INSERT Statement

```
$INSERT insert-file
```

Inserts into the program, at compilation time, the file whose pathname is insert-file. The $INSERT command cannot be nested: do not include a $INSERT command in a file which will be inserted into a program by a $INSERT command.

$INSERT is commonly used for:

- Insertion of COMMON specifications into programs
- Commonly used statement functions
- Data initialization statements
- Numeric key definitions, especially for the file management system, applications library, MIDAS, etc.

Note

Unlike other statements, the $INSERT statement must begin in Column 1.

ASSIGNMENT STATEMENTS

```
target = expression
```

target is any data item. expression is any expression whose type is or can be converted to that of the target.

When an assignment statement is executed, the expression is evaluated (unless it is just a single value) and the resulting value is assigned to the target. (See Section 2 for a discussion of expression evaluation.) Examples:

1. Arithmetic (A, B, and C are numeric variables)

```
A = B**2 + SIN(C)
A = 25.
```

2. Logical: (P, Q, R are logical variables).

```
P = .TRUE.
P = ((A.GT.B) .AND.  (B.GT.C))
```

3. Character: (C, D, E are character variables).

```
C = D // E // CHARFUNC (D//E)
```

Mixed-Type Assignments

When the target and the value being assigned to it are of differing types, the value will be converted, if possible, to the type of the target. The conversions differ for character, logical, and arithmetic data.

Arithmetic Conversions: Arithmetic targets can be assigned values of any arithmetic type. If the types differ, the compiler will automatically insert appropriate type-conversion routines (drawn from the set of intrinsic functions) into the code for the assignment statement. The conversions and assignments carried out in such cases are described in Table 3-1.

Logical Conversions: Type LOGICAL targets can be assigned only type LOGICAL values, but the storage lengths may differ. The value will be converted to the storage length of the target, then assigned.

Character Conversions: Type CHARACTER targets can be assigned only CHARACTER values, but the lengths need not match. The value will be truncated or blank-extended on the right so that it matches the length of the target, then assigned.

CONTROL STATEMENTS

DO Statement

```
DO s [,] i = m1, m2 [,m3]
```

where:

s is the statement number of the last statement in the range of the DO-loop.

i is an integer, real, or double precision variable.

m1, m2, and m3 are integer, real, or double precision expressions representing the initial value, the limit value, and the increment value respectively. If m3 is not specified, it is assumed to be one. It may not be zero.

```
DO 100 A = K+M,-(SQRT(Z)), -1.5
```

Execution: m1, m2, and m3 are evaluated prior to the first execution of the loop, and converted to the type of the index variable. The index takes on the values m1, m1 + m3, m1 + 2*m3, etc. The loop executes once following each assignment, until an assignment occurs such that

```
m1 + n*m3 > m2 for m3 > 0, or
m1 + n*m3 < m2 for m3 < 0.
```

When this occurs, the DO-test fails: control jumps immediately to

Table 3-1
Conversion Rules for Mixed-Type Assignments

Value Type	Target Type: I*2	I*4	REAL	DOUBLE	C*8	C*16
I*2	ASSIGN	EXTEND ASSIGN	FLOAT ASSIGN	DFLOAT ASSIGN	FLOAT ASREAL	DFLOAT ASREAL
I*4	TRUNC ASSIGN	ASSIGN	FLOAT ASSIGN	DFLOAT ASSIGN	FLOAT ASREAL	DFLOAT ASREAL
REAL	SFIX ASSIGN	LFIX ASSIGN	ASSIGN	DFLOAT ASSIGN	ASREAL	DFLOAT ASREAL
DOUBLE	SFIX ASSIGN	LFIX ASSIGN	FLOAT ASSIGN	ASSIGN	FLOAT ASREAL	ASREAL
C*8	SFIX* ASSIGN*	LFIX* ASSIGN*	ASSIGN*	DFLOAT* ASSIGN*	ASSIGN	DFLOAT ASSIGN
C*16	SFIX* ASSIGN*	LFIX* ASSIGN*	FLOAT* ASSIGN*	ASSIGN*	FLOAT ASSIGN	ASSIGN

Table 3-1 (continued)
Conversion Rules For Mixed-Type Assignments

Operation	Action
ASSIGN:	Transmit value (after any indicated conversion) to the target.
ASREAL:	ASSIGN value as above to the real part of a complex number, and set the imaginary part of the complex number to zero.
SFIX:	Truncate fractional part and convert result to a short integer. Overflow may occur.
LFIX:	Truncate fractional part and convert result to a long integer. Overflow may occur.
FLOAT:	Convert value to REAL form. Loss of precision may occur if the argument was DOUBLE PRECISION, COMPLEX*16, or INTEGER*4. Overflow may occur with DOUBLE PRECISION or COMPLEX*16.
DFLOAT:	Convert value to DOUBLE PRECISION form.
EXTEND:	Prefix the short integer with 16 binary Ø's or 1's if the short integer was positive or negative, respectively. This cannot change the value or sign of the integer.
TRUNC:	Discard the 16 high-order bits of the long integer. A value outside the short-integer range will be altered, and possibly changed in sign, by this operation.

An asterisk affixed to an operation involving a complex number indicates that the operation is to be performed on the real part only - the imaginary part is not involved. When no asterisk is present, the operation is to be performed on both parts of the number.

statement s and continues from there. The index variable will retain the value at which the test failed, and this value is available for use in subsequent program execution.

DO-loops may be nested. There is no syntactic limit to the nesting of DO-loops.

It is an undesirable programming technique to have the index variable appear as the initial, limit, or increment value in a DO statement. The index variable may not become redefined or undefined during execution of the range of the DO-loop. The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, arithmetic-IF, block-IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. The recommended practice is simply to end all DO-loops with a CONTINUE statement.

FTN Compatibility: The FORTRAN 77 DO-loop differs substantially from that in FTN. Errors, some undetectable by the F77 Compiler, can occur if these differences are ignored - particularly when FTN programs are converted to FORTRAN 77.

In FTN, control can leave and enter an active DO-loop using GO TO statements: this is called an "extended DO-range". In FORTRAN 77 it is illegal to branch into the body of a DO-loop at any time: no extended DO-range is permitted.

In FTN, the DO-test occurs after each execution of the loop; hence all DO-loops execute at least once. In FORTRAN 77 the DO-test occurs before execution. Hence, if the first test should fail (when i = m1 + Ø*m3) the loop will not execute at all.

In FTN, the index variable must be integral, the initial and limit values must be integer constants or variables (no expressions) and the increment must be positive. None of these restrictions apply to FORTRAN 77.

The F77 Compiler will generate a DO-loop of either type, depending on the option given in the command line. When the -DO1 option is given, all DO-loops produced will be of the FTN type, and the code for them must meet all the FTN restrictions. When the -NODO1 option (the default) is given, all DO-loops will be of the FORTRAN 77 type: the FTN restrictions are relaxed, and the prohibition of extended DO-ranges is enforced.

Caution

If an extended DO-range is present in a DO-loop compiled with -NODO1, the program will compile without errors, but will fail on execution. The cause of the failure may be very difficult to detect.

CONTINUE Statement

```
[statement-label]  CONTINUE
```

The CONTINUE statement serves as a point of reference in a FORTRAN 77 program: it is a peg on which to hang a label. It is usually used to indicate the end of the range of a DO-loop.

ASSIGN Statement

```
ASSIGN k TO i
```

Where i is an integer variable, and k is a statement label. An ASSIGN statement must be executed prior to an assigned GO TO. Once i has been ASSIGNed, it may be used only in an assigned GO TO until it has been given an integer value by an arithmetic assignment.

Unconditional GO TO Statement

```
GO TO k
```

Transfers control to statement labeled k.

Computed GO TO Statement

```
GO TO  (k [,k]...) [,] i
```

Transfers control to the statement whose label is in the n'th position in the list of k's when integer expression i = n. If there is no n'th statement label, control passes to the next executable statement after the computed GO TO.

Assigned GO TO Statement

```
GO TO i [[,] (k [,k]...)]
```

Where i is an integer variable, and each k is the label of an executable statement in the program unit containing the assigned GO TO. Transfers control to the statement labeled i. Prior to executing the assigned GO TO, a statement label value must be assigned to i using the ASSIGN statement. The list of k's is optional. If it appears, the statement label assigned to i must be one of the labels in the list.

There is no syntactic limit to the number of labels in a computed or assigned GO TO.

Arithmetic-IF Statement

```
IF (e) k1, k2, k3
```

Where e is an arithmetic expression with an integer, real, or double precision value. If e < Ø (negative) control is transferred to statement labeled k1; if e = Ø (exactly), control is transferred to statement labeled k2; and if e > Ø (positive), control is transferred to statement labeled k3.

The arithmetic-IF is obsolete: use the block-IF statement in new programs.

Logical-IF Statement

```
IF (e) statement
```

Where e is a logical expression and statement is any valid executable statement except a DO, logical-IF, block-IF, ELSE IF, ELSE, or END IF statement. If e is true, the statement is executed; if e is false, control passes to the next executable statement.

Note

An arithmetic-IF may be the statement in a logical-IF but this is not recommended as a good programming practice.

Block-IF Statement

```
IF (logical expression) THEN
[statements]
[ELSE IF (logical expression) THEN ] ...
[[statements]                      ]
[ELSE          ]
[[statements]  ]
END IF
```

Allows a block of statements to be executed if an associated logical expression is true, or skipped if it is false. Scans a series of such blocks, executes the first whose expression is true, and skips over the remaining blocks automatically.

There may be any number of ELSE IF statements, or none. There may be at most one ELSE statement, which must follow any ELSE IF statements. The blocks may contain any number of statements, or none.

Execution: The logical expressions in the IF and any ELSE IF statements are evaluated in the order they appear. If an expression is false, the next expression is evaluated. If an expression is true, the block of statements between it and the next ELSE IF, ELSE, or END IF statement is executed. Control then jumps (or passes) to the END IF statement and proceeds sequentially from there. If no expression is

true and an ELSE statement is present, the block of statements following it is executed. Otherwise, none of the blocks is executed, and control proceeds from the END IF statement.

Nesting: Any of the statements controlled by a block-IF can be another block-IF. The ELSE IF, ELSE, and END IF statements of a nested block-IF are local, and do not affect the flow of control in the containing block-IF. Nested block-IF's should be indented to indicate this independence.

Note

Transfer of control into a block-IF from outside is prohibited. Entry may occur only through the initial IF statement.

When a DO-loop is present in a block-IF, it must be wholly contained in the statement block in which it begins. When a block-IF is present in a DO-loop, it must be wholly contained in the body of the loop.

PAUSE Statement

```
PAUSE [n]
```

Where n is an optional decimal number of up to five digits or a CHARACTER constant. Halts the program and prints ****PAUSE n at the keyboard. Keying in START continues operation of the program at the next executable statement following the PAUSE.

RETURN Statement

```
RETURN [n]
```

Used in a subprogram to cause return to the calling program. Any number of RETURN statements may be present. A RETURN should appear before the END statement, but one will be assumed if it does not.

In a subroutine, the integer expression n may be specified. Execution of RETURN n causes return to the statement of the calling program unit whose label was passed as the n'th statement-label dummy argument in the subroutine argument list. See Section 5 for details.

STOP Statement

```
STOP [n]
```

Where n is an optional decimal number of up to five digits or a CHARACTER constant. Halts program execution, closes all file units referenced by the program, prints ****STOP n at the keyboard, and returns control to the PRIMOS level. A STOP statement may appear anywhere in a program unit. In a main program, an END without a STOP

causes a STOP to occur automatically. If the effect of a STOP without the printing of '****STOP', is desired, use CALL EXIT rather than STOP.

END Statement

END

The final statement of a program, subroutine (including a BLOCK DATA subroutine) or external function. Tells the compiler that it has reached the physical end of the program unit. In a main program, END implies STOP if no STOP statement precedes it. In a subprogram, END implies RETURN if no RETURN statement precedes it.

SUMMARY OF STATEMENT SYNTAX

In the following table, all program specification statements are listed in alphabetical order with their syntax requirements. The table is intended only as a reminder for those already familiar with the statements.

Table 3-2.

Specification Statement Syntax

Statement	Syntax
Arithmetic-IF	IF (e) k1, k2, k3
ASSIGN	ASSIGN k TO i
Assigned GO TO	GO TO i [[,] (k [,k]...)]
BLOCK DATA	BLOCK DATA [name]

Block-IF	IF (logical expression) THEN
	[statements]
	ELSE IF (logical expression) THEN ...
	[statements]
	ELSE
	[statements]
	END IF

CALL	CALL name [([argument [,argument]...])]
COMMON	COMMON /x/a [,/x/a]... (Commas are optional)
Computed GO TO	GO TO (k [,k]...) [,] i
CONTINUE	[statement-label] CONTINUE
DATA	DATA k/d/ [,k/d/]... (Commas are optional)
DIMENSION	DIMENSION array declarator [,array declarator]...
DO	DO s [,] i = m1, m2 [,m3]
END	END
ENTRY	ENTRY entry-name [([argument [,argument]...])]
EQUIVALENCE	EQUIVALENCE (k ,k [,k]...) [,(k ,k [,k]...)]...
EXTERNAL	EXTERNAL name [,name]...
FUNCTION	[type] FUNCTION name ([argument [,argument]...])
GO TO	GO TO k
IMPLICIT	IMPLICIT type (list) [,type (list)]...

Table 3-2 (continued)

Specification Statement Syntax

Statement	Syntax
INSERT	$INSERT insert-file (Must start in Col. 1)
INTRINSIC	INTRINSIC name [,name]...
LIST	LIST
Logical-IF	IF (e) statement
NO LIST	NO LIST
PARAMETER	PARAMETER (p=c [,p=c]...)
PAUSE	PAUSE [n]
PROGRAM	PROGRAM name
RETURN	RETURN [n]
SAVE	SAVE [v [,v]...]
STOP	STOP [n]
SUBROUTINE	SUBROUTINE name [([argument [,argument]...])]
Type-statement	type v [,v]...

SECTION 4

INPUT/OUTPUT STATEMENTS

The following brief discussion is intended only as a review, to establish the context in which FORTRAN I/O commands operate. Those unfamiliar with the features mentioned should consult an appropriate textbook.

Input/Output in FORTRAN 77 is based on logical records stored in files. The physical aspects of record and file storage are not dealt with by the language. Hence, the following descriptions are concerned only with the logical structures involved.

F77 DATA STORAGE

A file is a sequence of bytes stored in or accessible to the computer. A file may be empty, or may contain one or more records. The records are subsequences of bytes separated from each other either physically or logically (or both) in such a way that they can be read or written individually. The record is the basic unit of data transfer.

Every open file has a pointer. When a file is first opened, its pointer is positioned before the first record. For data transfer, the pointer first moves to the beginning of the selected record (direct access) or the next record in the file (sequential access), then sweeps across the record as the record is read or written. After data transfer, the pointer remains at the end of the record just read or written, or after the endfile record if one was written or encountered.

In an interactive environment, the sequence of bytes stored in the user terminal is considered a file, since it has all the qualities usually associated with a file except permanence. Usually no distinction is needed between the terminal and the file it stores: one just speaks of writing to or reading the terminal. Do not confuse the terminal's cursor with its file pointer.

Types of Record

There are three types of record: formatted, unformatted, and endfile. No file may contain both formatted and unformatted records.

Formatted Record: A formatted record consists entirely of ASCII characters. Such a record can be accessed only in conjunction with a format list, which tells the computer how to translate the ASCII data to or from representations suitable for internal processing.

Unformatted Record: An unformatted record contains data in the same form in which it is actually used by the computer. No format list is used when it is accessed: the data is transcribed directly to or from the storage medium.

Endfile Record: An endfile record is written by an ENDFILE statement. It may occur only as the last record of a sequential file. Encounter of the endfile record during a READ informs the system that the file has been exhausted. See ENDFILE Statement, below.

Record Lengths

The length of a record is measured in bytes. In a formatted record, each byte holds one character.

Formatted and unformatted records may be stored either in fixed-length form or varying-length form. No file may contain both fixed- and varying-length records.

Fixed Length: A file of fixed-length records is produced when the RECL (record length) option is given in the OPEN statement creating the file. All records written to the file will be of the length specified.

Use of the RECL option for a sequential-access file is an F77 extension.

Varying Length: A file of varying-length records is produced when the RECL option is omitted from the OPEN statement creating the file. Each record will have the length needed to hold its data, up to the current maximum record length. See INCREASING MAXIMUM RECORD LENGTH, below. Files of varying-length records cannot be used under direct access.

Implementation: The following information is significant only when space conservation on disc files is a major concern.

Files of varying-length records are kept in compressed ASCII format: sequences of identical characters, most often blanks, are replaced by one example of the character and a repeat count. All such files are processed through Physical Device 7. Compressed format is maximally economical of space, but requires some additional processing time to compress and uncompress the records.

Files of fixed-length records are kept in uncompressed ASCII format: records are stored just as they are created by the program, with no compression. All such files are processed through Physical Device 8. Uncompressed format can be quite wasteful of space, but I/O on uncompressed files is faster than on compressed files because no time is spent compressing and decompressing the records.

For further information, see The PRIMOS Subroutines Reference Guide.

Types of File Access

There are two types of file access: sequential and direct.

Sequential Access: Under sequential access, the file pointer can move only one record at a time, either forward to transfer data, or backward due to a BACKSPACE command; or it may jump to the beginning of the file (REWIND).

Direct Access: Under direct access, the file pointer can jump to the beginning of any record in the file and read or write it. BACKSPACE, ENDFILE, and REWIND are not applicable to direct access.

Types of File

There are two types of file in F77: those which were created under the sequential access method - SAM files - and those created under direct access - DAM files. The two file organizations are quite different. A special type of sequential file, the internal file, is discussed separately.

SAM Files: In a SAM file, the records are stored in the order they were written, and are usually read in that order. New records can be added only to the end of the file, and records cannot be deleted. SAM files can be read or written only under sequential access. SAM records may be of fixed- or varying-length.

DAM Files: In a DAM file, the records are stored as required to facilitate direct access - the details are beyond the scope of this book. New records can be added anywhere in any order; existing records can be deleted by over-writing them. DAM files must be written only under direct access, but can be read by either direct or sequential access. DAM records must always be fixed-length.

Every record in a DAM file is identified by a key (a positive integer). This key is specified when the record is written. Under direct access, a record is retrieved by giving its key in a direct access READ statement. Under sequential access, a DAM file acts like a SAM file to which the records were written in order by key; a record is retrieved by reading through the file until it is reached.

Caution

> A direct access file must not be modified by the Editor or any sequential data transfer statement, or its usability for direct access will be partly or wholly lost.

Internal Files

These provide a way to convert data from one form to another within main memory.

An internal file is an area of memory where a type CHARACTER variable, array, array element, or substring is stored. Such an area acts as an internal file when the name of the data item stored there is given in place of the file unit number in a formatted, sequential READ or WRITE statement.

The READ or WRITE proceeds as usual, but the "file" used is the designated internal storage area, rather than an external file on secondary storage. Data is transferred to or from the file area, after conversion as directed by the associated format list.

An internal file contains only one record. After each read or write, its pointer returns automatically to the beginning of the record.

The ENCODE and DECODE statements of FTN are obsolete in FORTRAN 77. Specify an internal file to an ordinary READ or WRITE, as described.

EDITING F77 FILES

The PRIMOS Editor produces and expects SAM files of formatted, varying-length records. A file created with these attributes by an F77 program may be edited freely. A file created by the Editor may be opened with these attributes in an F77 program and modified as desired.

The Editor should not be used to modify a SAM file of fixed-length, formatted records, because it will automatically compress the file, effectively transforming it to a file of varying-length records. Neither should it be used on a DAM file, since it will not maintain the fixed-length records a DAM file requires. The editor may be used to examine a fixed-length file provided it is not refiled (no FILE command given).

The Editor cannot process unformatted files.

INCREASING MAXIMUM RECORD LENGTH

When the shared libraries are used in loading an F77 program (unqualified LI command to SEG during loading) records of all types have a maximum length of 32K bytes. This limit cannot be increased.

When the unshared libraries are used (LI NPFTNLB and LI IFTNLB to SEG during loading) the maximum record size is initially 256 bytes, but it may be increased to up to 32K bytes. When records longer than 256 bytes are needed, the PRIMOS I/O Control System (IOCS) must be notified. Two aspects of IOCS are involved:

- The size specified by the variable in the I/O size-control block F$IOSZ.
- The size of the I/O buffer F$IOBF. This buffer is discussed further under Data Transfer Statements, below.

Specifications to IOCS must be given in two-byte words. To increase the maximum record length, proceed as follows:

1. Increase the value specified in F$IOSZ to the desired record length by inserting the following statements into the main program:

```
COMMON/F$IOSZ/MAXSIZE
INTEGER*2 MAXSIZE/halfwords/
```

   where halfwords is an integer constant giving the desired record length in two-byte words (half the length in bytes).

2. Increase the size of F$IOBF to the desired record length by inserting the following statements into the main program:

```
COMMON/F$IOBF/BUFSIZE
INTEGER*2 BUFSIZE (halfwords)
```

   where halfwords is as above.

Any variable names could be used in place of MAXSIZE and BUFSIZE.

The reason no special action is needed to obtain the maximum record size when using the shared libraries is that they automatically provide a MAXSIZE and BUFSIZE of 16K words (32K bytes).

Note

The value in F$IOSZ and the size of F$IOBF set an upper size limit on all records, but do not determine the actual record size for any particular file. The actual record size for a fixed-length file is determined by the RECL option in the OPEN statement for the file. (See below.) Arguments to RECL must be given in bytes. For varying-length files, including the terminal, it depends on the individual record.

FILES AND PROGRAMS

Before a program can read or write a file, the programmer must establish a connection between the file and the program. This is accomplished by assigning a device if necessary, and by opening the file on a file unit.

Assigning a Device

When a file is on the card punch or reader, the paper tape punch or reader, a magnetic tape drive, or is being written directly to the line printer without the use of SPOOL, the device must be ASSIGNed before program execution begins. See The Prime User's Guide.

Opening a File on a File Unit

A file unit is a numbered channel through which data flows between a program and a file. Every file except the user terminal, which is always open on file unit 1, must be connected to a file unit prior to data transfer. There are three ways of connecting a file to a file unit:

- With the FORTRAN 77 OPEN statement. This is the usual way.
- With a call to one of the PRIMOS file-opening subroutines. These provide more power and flexibility than the FORTRAN 77 OPEN, but these advantages are usually not needed.
- With a PRIMOS OPEN command executed before the program is run. This is known as preconnection.

A preconnected file may be opened again within the program, and additional attributes added to the connection. (See OPEN Statement below). In case of conflicting attributes, those specified within the program take precedence.

See The PRIMOS Subroutines Reference Guide for details on the PRIMOS file-opening subroutines.

Caution

PRIMOS and F77 use different numbering systems to describe the set of file units. When a file unit is referenced in F77, its FORTRAN unit number must be used. When it is referenced in a PRIMOS subroutine call, the corresponding PRIMOS Funit number must be given instead. Beware of confusing the two descriptive systems. See Table 4-1.

Integer arguments to most PRIMOS Subroutines must be INTEGER*2.

Table 4-1. Devices and Their Default FORTRAN Unit Numbers

FORTRAN Unit Number	PRIMOS DEVICE
1	User terminal
2	Paper tape reader or punch
3	MPC card reader
4	Serial line printer
5	Funit 1
6	Funit 2
7	Funit 3
8	Funit 4
9	Funit 5
10	Funit 6
11	Funit 7
12	Funit 8
13	Funit 9
14	Funit 10
15	Funit 11
16	Funit 12
17	Funit 13
18	Funit 14
19	Funit 15
20	Funit 16
21	9-track magnetic tape unit 0
22	9-track magnetic tape unit 1
23	9-track magnetic tape unit 2
24	9-track magnetic tape unit 3
25	7-track magnetic tape unit 0
26	7-track magnetic tape unit 1
27	7-track magnetic tape unit 2
28	7-track magnetic tape unit 3
29	Funit 17
.	.
.	.
.	.
.	.
.	.
139	Funit 127

The mapping of FORTRAN unit numbers to PRIMOS Devices shown here may be altered for the duration of a program through a call to the PRIMOS Subroutine ATTDEV. See The PRIMOS Subroutine Reference Guide.

FILE OPERATIONS

The possible operations on a file are:

- Create a new file (OPEN).
- Access an old file (OPEN).
- Change file-connection attributes (OPEN).
- Determine current status and attributes of a file (INQUIRE).
- Transfer data to/from a file (READ, WRITE, PRINT, FORMAT).
- Indicate the end of a file (ENDFILE).
- Reposition the file pointer (BACKSPACE, REWIND).
- Disconnect from a file (CLOSE).
- Delete a file (Options in OPEN and CLOSE).

The statements which perform these operations are divided into four categories:

- File Control Statements:

  OPEN
  CLOSE
  INQUIRE

- Device Control Statements:

  ENDFILE
  BACKSPACE
  REWIND

- Data Transfer Statements:

  READ
  WRITE
  PRINT

- Format Statement:

  FORMAT

FILE CONTROL STATEMENTS

File control statements establish, alter, or read out the current attributes and status of a file. In file control statements, all integer arguments must be INTEGER*4 and all logical arguments must be LOGICAL*4. An argument that is not an expression may be either a variable or an array element.

OPEN Statement

```
OPEN ([UNIT= ]unit# [,FILE= filename] [,STATUS= stat] [,ACCESS=acc]
     [,FORM= fm] [,RECL= reclngth] [,BLANK= blnk] [,ERR= label]
     [,IOSTAT= ios])
```

An OPEN statement may be used to create a new file and establish its basic properties, or to connect a file to a file unit and establish the properties of the connection. For a new file, one OPEN statement will perform both these functions. The same file may be connected with different properties at different times, but must always be closed before it is reopened.

The options used may be given in any order, except that if UNIT= is omitted, unit# must appear first. The meanings of the options, and the data types required for the arguments, are described in Table 4-2.

The following is an example of the OPEN statement:

```
INTEGER*4  STATVAL
CHARACTER*2Ø  ACCTYPE
ACCTYPE = 'SEQUENTIAL'
OPEN (1Ø, FILE= 'YORD', STATUS= 'OLD', ACCESS= ACCTYPE,
      FORM= 'FORMATTED', RECL= 25, ERR= 999, IOSTAT= STATVAL)
```

An existing file named YORD is opened for formatted sequential access on file unit 1Ø. The record length is 25. Should a numeric field containing blanks be read from the file, the blanks will be deleted. Should an error occur, for instance if the file does not in fact exist, or unit 1Ø is already in use, control will transfer to Statement 999, and STATVAL will be given a positive value.

PRIMOS File-Opening Subroutines: These permit files to be created interactively at run time, allow files to be opened with various protection attributes, and provide other services additional to those of the FORTRAN 77 OPEN Statement. See The PRIMOS Subroutines Reference Guide. See also the Caution under FILES AND PROGRAMS, above.

Table 4-2. OPEN Statement Options

Option	Argument Data-Type	Results of Arguments Specified
UNIT=	Integer*4 Expression	File is opened on the file-unit specified.
FILE=	Character Expression	The file has the name specified. A pathname may be used. If no FILE= is specified for a new non-scratch file, the file will be named F#nnn where nnn is the number of the file-unit on which the file was opened.
STATUS=	Character Expression	'OLD': Specified if the file already exists. 'NEW': Specified if the file is being created. 'SCRATCH': File is temporary: it will be automatically deleted at program end. No file name may be specified. 'UNKNOWN': (Default) Specified if the status is not known to the programmer. The processor will determine the appropriate status.
ACCESS=	Character Expression	'SEQUENTIAL': (Default) File is connected for sequential access. 'DIRECT': File is connected for direct access.
FORM=	Character Expression	'FORMATTED': (Default under sequential access) File is connected for formatted data transfer. 'UNFORMATTED': (Default under direct access) File is connected for unformatted data transfer.
RECL=	Integer*4 Expression	Sets record length for a file of fixed-length records. Must be omitted for a file of varying-length records. Use in SAM files is an F77 extension.

Required in DAM files.

BLANK=	Character Expression	This item specifies treatment of blanks in numeric input fields when data is read into the file. 'NULL': (Default) All blanks are deleted, and digits compressed to the right side of the input field. An all-blank field will be interpreted as a zero value. 'ZERO': All but leading blanks are converted to zeroes, as in FORTRAN 66.
ERR=	Statement Label	Control transfers to statement specified if an error occurs during execution of the OPEN statement.
IOSTAT=	Integer*4 Variable	Set to zero if the OPEN statement executes successfully. Set positive on error in OPEN-statement execution.

CLOSE Statement

```
CLOSE ([UNIT= ]unit# [,STATUS= stat] [,ERR= label] [,IOSTAT= ios])
```

The CLOSE statement disconnects a file from a unit. ERR= and IOSTAT= have the same significances as in the OPEN statement. STATUS= determines the final disposition of the file. The argument stat is a character expression which may have the values:

'KEEP' The file will be retained after it is closed. This is the default for non-SCRATCH files, and must not be given for SCRATCH files.

'DELETE' The file will be deleted after it is closed. Default for SCRATCH files.

The options used may be given in any order, except that if UNIT= is omitted, unit# must appear first.

When execution terminates normally, all files opened or referenced in the program (except COMO files) are automatically closed. However, when execution terminates due to an error, all open files remain open.

INQUIRE Statement

```
INQUIRE ([FILE= ]filename or [UNIT= ]unit# [,IOSTAT= ios]
         [,ERR= s] [,EXIST= ex] [,OPENED= od] [,NUMBER= num]
         [,NAMED= nmd] [,NAME= fn] [,ACCESS= acc]
         [,SEQUENTIAL= seq] [,DIRECT= dir] [,FORM= fm]
         [,FORMATTED= fmt] [,UNFORMATTED= unf] [,RECL= rcl]
         [,NEXTREC= nr] [,BLANK= blnk])
```

Where filename is a character expression, and unit# is an integer expression.

An INQUIRE statement is used to ascertain the properties of a file, or of its connection to a file unit. Each option acts as a question: when the INQUIRE statement executes, the variable supplied by the programmer for each option is set to a value that answers the question the option asks. The correct data types for the variables, and the meanings of the various responses, are described in Table 4-3.

The file must be specified by name (INQUIRE by name) or unit (INQUIRE by unit) but not both. Options may appear in any order, but no option may appear more than once. If FILE= (or UNIT=) is omitted, the filename (or unit#) must appear first.

A variable or array element that may become defined or undefined as a result of its use in an INQUIRE statement, or any associated data item, must not be referenced by any other option in the same INQUIRE statement.

Table 4-3. INQUIRE Statement Options

Specifier	Argument Data Type	Significance of Possible Values
FILE=	Character Expression	Specifies file by name.
UNIT=	Integer*4 Expression	Specifies file by unit number.
IOSTAT=	Integer*4	Zero: no error condition exists.
		Positive: error condition exists.
ERR=	Statement number	Control transfers to statement indicated if error occurs during INQUIRE statement execution.
EXIST=	Logical*4	.TRUE.: the file exists (for INQUIRE by name) or the unit exists (for INQUIRE by unit).
		.FALSE.: the file or the unit does not exist.
OPENED=	Logical*4	.TRUE.: the file is open (INQUIRE by name) or the file unit is open (INQUIRE by unit).
		.FALSE.: the file or the unit is not open.
NUMBER=	Integer*4	Variable supplied is set to the file's unit-number. If there is none, variable becomes undefined.
NAMED=	Logical*4	.TRUE.: the file has a name.
		.FALSE.: the unit has no name.
NAME=	Character	Variable is set to the file name. If none or file not connected, variable becomes undefined.
ACCESS=	Character	'SEQUENTIAL': file open for sequential access.
		'DIRECT': file open for direct access.
		Becomes undefined if file is closed.

Table 4-3. INQUIRE Statement Options (continued)

SEQUENTIAL=	Character	'YES': file can be connected for sequential access. 'NO': file cannot be connected for sequential access. 'UNKNOWN': suitability of the file for sequential access cannot be determined.
DIRECT=	Character	'YES': file can be connected for direct access. 'NO': file cannot be connected for direct access. 'UNKNOWN': suitability of file for direct access cannot be determined.
FORM=	Character	'FORMATTED': open for formatted data transfer. 'UNFORMATTED': open for unformatted data transfer. Becomes undefined if file is not open.
FORMATTED=	Character	'YES': file consists of formatted records. 'NO': file consists of unformatted records. 'UNKNOWN': record type cannot be determined.
UNFORMATTED=	Character	'YES': file consists of unformatted records. 'NO': file consists of formatted records. 'UNKNOWN': record type cannot be determined.

Table 4-3. INQUIRE Statement Options (continued)

RECL=	Integer*4	Variable is set to the record-length for which the file is open. Becomes undefined if file consists of varying-length records or is closed.
NEXTREC=	Integer*4	Variable is assigned the value n+1 where n is the record number of the last record read or written on a file connected for direct access. If no records have been read or written, the variable is set to 1. If the file is not connected for direct access, or if the position of the file pointer is indeterminate due to a previous error, the variable becomes undefined.
BLANK=	Character	'ZERO': non-leading blanks in numeric fields will be converted to zeroes. 'NULL': non-leading blanks in numeric fields will be deleted. If the file is not open for formatted data transfer, the variable becomes undefined.

DEVICE CONTROL STATEMENTS

These statements apply only to sequential (SAM) files. They reposition the file pointer, either physically (file on tape) or logically (file on disc), or write the endfile record that prevents a device from reading off the end of a file.

'IOSTAT=' and 'ERR=' are as described for the OPEN statement. If only unit# is specified, the parentheses may be omitted.

BACKSPACE Statement

BACKSPACE ([UNIT=]unit# [,IOSTAT= ios] [,ERR= label])

Moves the pointer of a file open for sequential access back to the beginning of the previous record. BACKSPACE may be performed:

- On any formatted file, except that records written using list-directed I/O may not be backspaced over.
- On any unformatted file having a fixed record-length (RECL size specified in the OPEN statement).

When a file pointer is positioned after the endfile record, as is the case after the ENDFILE condition has been raised, BACKSPACE will reposition the file pointer before the endfile record. When a file pointer is at the initial point of the file, BACKSPACE has no effect.

REWIND Statement

REWIND ([UNIT=]unit# [,IOSTAT= ios] [,ERR= label])

Repositions the file pointer to the initial point of a file, either by physically rewinding a tape, or by resetting a disc file's logical pointer. When a file pointer is at the initial point of the file, REWIND has no effect.

ENDFILE Statement

ENDFILE ([UNIT=]unit# [,IOSTAT= ios] [,ERR= label])

Writes a device-specific endfile record on the file connected to the file unit unit#. The pointer is left positioned after the endfile record. This statement can also be used to truncate disk files.

On a sequential tape file, an endfile record must be explicitly written following the last data record.

On a sequential disc file, the computer will supply an endfile record automatically whenever one is appropriate. However, use of an explicit ENDFILE statement for such files is strongly recommended, for compatibility with other systems.

On a DAM file, no endfile record should ever be written. If one is, unpredictable and undesirable results will occur.

DATA TRANSFER STATEMENTS

These control the actual transfer of data between files and program variables. READ moves data out of files. WRITE and PRINT move data into files.

Data is not transferred directly between files and program variables. In a READ, the current record is first transferred from the file to the FORTRAN I/O buffer F$IOBF, which resides in main memory. The FORTRAN I/O system then scans F$IOBF (using a pointer similar to a file pointer), reads out the separate data items, edits them if the READ is formatted, and assigns them to the appropriate variables. In a WRITE, the order is reversed: the data items are edited or transferred into F$IOBF, then the contents of F$IOBF are written as a whole to the file.

Usually F$IOBF is scanned sequentially. However, the T edit-control descriptor can be used in a formatted data transfer to scan it in any desired order: see under Edit-Control Descriptors, below.

For simplicity, the following descriptions will not mention F$IOBF, since the programmer need not be concerned with it except when its size must be increased (See INCREASING MAXIMUM RECORD LENGTH, above) or the T descriptor is used.

Data transfer statements may be used to convert data from one type to another inside the computer. See Internal Files, above.

F77 accepts both the ANS and IBM formats in direct-access READs and WRITEs. Details of these formats are given with the individual statements.

Note

A function must not be referenced anywhere in a data transfer statement if the reference causes execution of a data transfer statement.

READ Statement

Sequential: READ ([UNIT=]unit# [,[FMT=]format] [,END= label] [,ERR= label] [,IOSTAT= ios]) [input list]

ANS direct: READ ([UNIT=]unit# [,[FMT=]format] ,REC= record# [,END=label] [,ERR=label] [,IOSTAT=ios]) [input list]

IBM direct: READ (unit#'record# [,[FMT=]format] [,END= label] [,ERR= label] [,IOSTAT= ios]) [input list]

The unit# is an integer expression specifying the FORTRAN file unit to be read. It must be present. All other items are optional. An asterisk may be given for unit#: this is equivalent to specifying file unit 1, the terminal.

If a format is present, the read is formatted; otherwise it is unformatted. A format may be any of the following:

- The statement number of a FORMAT statement
- An INTEGER variable that has been ASSIGNed such a number
- A CHARACTER array name, array element, variable, or constant
- A fixed-length CHARACTER expression
- An asterisk, denoting list-directed I/O

When a format consists of any character entity, the entity must contain the same format list, including outer parentheses, that would appear following the keyword FORMAT in an ordinary FORMAT statement. Only those positions which will actually be referenced during data transfer need be defined. Any data at other positions will be ignored. If an unsubscripted array is used, the format list will be obtained from the concatenation of all its elements. Blanks are of no significance in any type of format list.

A record# is an integer expression. If a record# is present, the READ statement is a direct-access READ; otherwise it is a sequential-access READ. Any file may be read sequentially, but only a file created for direct access (DAM file) can be read by direct access.

If END= label appears, control will transfer to the statement label specified by label (an integer constant) if endfile should occur during the READ. Do not specify END= for a direct-access read.

If ERR= label appears, control will transfer to label if an error should occur during the READ.

If IOSTAT= ios appears, ios (an integer variable) will be set to:

- A positive value if an error occurred
- Zero if the READ executed successfully
- A negative value if endfile was encountered and no error occurred

Note

In an IBM direct READ, unit#'record# must be the first item in the list.

If UNIT= is omitted from a sequential or ANS direct READ, unit# must be the first item in the list.

If FMT= is omitted from any formatted READ, format must be the second item in the list, and UNIT= must not appear.

In all other cases, the items may appear in any order.

Input Lists: An input list is a list of variables, arrays, array elements, and character substrings. These data items provide the destination of the data transferred in a READ statement. An input list may be empty, in which case the record is read but skipped. Redundant parentheses may not appear in an input list.

Input lists may contain implied-DO loops, to simplify assignment of data to arrays. An implied-DO follows the same rules as an ordinary DO. The DO-loop control values may have been read in at an earlier stage of the READ statement. Implied-DO loops may be nested; for each implied-DO, a set of parentheses must exist surrounding it, the array names it references, and any DO-loops nested within it. An implied-DO must be preceded by a comma.

Array elements not specifically referenced in a READ remain unchanged. If an array name appears without indexes, the computer will generate implied-DO loops to scan it in storage order. Assumed-size dummy arrays may not appear in input lists.

Input list examples:

```
DIMENSION ARR(-1:1Ø,-1:1Ø), VEC(5)

READ(1,1ØØ) L, M, ARR(L,M), N, VEC(N)

READ(1,2ØØ) ARR                                  }
                                                  } Equivalent
READ(1,3ØØ) ((ARR(I,J), I=1,1Ø), J=1,1Ø)          }

READ(1,4ØØ) J, K, M, N, (((ARR(I,L), L=M,N) I=J,K)
```

WRITE Statement

Sequential: WRITE ([UNIT=]unit# [,[FMT=]format] [,ERR= label] [,IOSTAT= ios]) [output list]

ANS direct: WRITE ([UNIT=]unit# [,[FMT=]format] ,REC= record# [,ERR= label] [,IOSTAT= ios]) [output list]

IBM direct: WRITE (unit#'record# [,[FMT=]format] [,ERR= label] [,IOSTAT= ios]) [output list]

WRITE Statements differ from READ Statements primarily in the direction of data transfer. The unit#, format, record#, ERR=, and IOSTAT= specifiers have the same significance as in a READ Statement. END= is not an option, and ios will never become negative, because endfile cannot occur when a file is written.

The rules governing omission of UNIT= and FMT= are the same as for a READ statement. See the Note above.

Output lists: An output list has the same form as an input list. The data items in an output list provide the source of the data transferred in a WRITE statement. They must all be defined when the WRITE occurs. An output list may be empty, in which case a null record is written.

Output lists may contain implied-DO loops and array names without indexes, which act as they do in input lists. They may also contain expressions. Any CHARACTER expression in an output list must be fixed-length. When the WRITE statement executes, each expression is evaluated and the result written to the file. An expression might consist only of a constant, in which case the constant is written. A format descriptor for an expression must be appropriate to the data type of its final value. If an output list expression contains function references, invocation of the functions must not change any other value in the expression, either directly or indirectly.

Length Mismatch: When a fixed-length record is written, the output list need not always have the same byte-length as the record. When an attempt is made to write a record too short to hold all the output list items, an error will occur. When a record longer than necessary to hold the output list is written, the extra positions will be padded with blanks if the WRITE is formatted, or with binary zeroes if the WRITE is unformatted. Padding of extra positions in unformatted DAM file records is an F77 extension; FORTRAN 77 leaves such positions undefined.

Carriage Control: The first character of each record in a file to be printed controls vertical spacing, and is not printed. The remaining characters in a record are printed starting at the left-hand margin. The significance of the permissible carriage-control characters is:

Character	Vertical Spacing Before Printing
Blank	One line
Ø (zero)	Two lines
1	To first line of next page
+	No advance (overprint of last line)

Records that contain no characters, generated by slash editing in a FORMAT Statement or by an empty output list, cause a blank line to be printed.

Unrepresentable Values: If a numeric item cannot be printed in the form required by a format code, the output field will be filled with asterisks.

PRINT Statement

```
PRINT format [,output list]
```

PRINT is a simplified WRITE. It prints the output list at the user terminal according to the format given in format. The format is as described for READ and WRITE. Equivalent to:

```
WRITE (1,format) [output-list]
```

For error handling (see below), a PRINT acts as a WRITE in which no options were given.

LIST-DIRECTED I/O

Also known as format-free I/O, list-directed I/O occurs when an asterisk appears as the format in a READ, WRITE, or PRINT Statement.

When list-directed output occurs, the values in the output list are converted to printable form as directed by FORTRAN-supplied format list defaults. The values are then written to the designated file.

List-directed input is usually employed when data is being read by a program from a free-format device such as the user terminal. A data item for list-directed input must have the same form as a constant of its data type. (See Section 2.) FORTRAN 77 supplies default format descriptors appropriate to the types of the data items in the input list, and uses those descriptors to convert the data as it is read in. List-directed I/O cannot be used in accessing internal files or DAM files.

Additionally, this feature provides a method to indicate in the input data that an item in the input list is to remain unchanged by a READ Statement. This is accomplished by using appropriate delimiters.

Delimiters

Adjacent values in a data line for list-directed input must be separated by one or more blanks, a comma, or a slash. Consecutive blanks are equivalent to single blanks. Blanks adjacent to a comma or slash are of no significance. An end-of-record is treated as a blank.

Two adjacent commas with no intervening characters except blanks will leave the corresponding item in the input list unchanged. A slash terminates a read, leaving any remaining items in the input list unchanged. A list-directed READ continues until a slash is encountered or all the items in the input list have been satisfied. If there are not enough values to complete the READ, an error will occur unless the data is being read from the terminal, in which case the program will wait for the remaining values to be typed in.

Repeat Counts

Repeat counts may modify data items under list-directed input.

```
r*c
```

represents r consecutive occurrences of the input value c. If c is omitted, r null values are read in, leaving the next r elements of the input list unchanged. No blanks may appear between r, *, and c.

Examples:

1. Source line: `READ(1,*) A,B,C,D`
   Input data: `151,,2*2E2`
   Result:
   ```
   A = 151.
   B is unchanged
   C = 2.E2
   D = 2.E2
   ```

2. Source line: `READ(1,*) I,J,K`
   Input data: `5 -3 /`
   Result:
   ```
   I - 5
   J = -3
   K is unchanged
   ```

INPUT/OUTPUT ERRORS

If an error occurs during execution of a READ or WRITE (including PRINT), execution of the statement terminates and the position of the file pointer becomes indeterminate.

If an error or endfile condition occurs during a READ statement, the data items in the input list and any implied-DO index variables become undefined. Data items used solely in subscripts, substring expressions, and implied-DO control values do not become undefined.

If an error occurs during a WRITE statement, any implied-DO index variables become undefined. The contents of the file remain as they were before execution of the WRITE began.

If an error occurs during a READ or WRITE that contains no IOSTAT= or ERR= option, or if endfile occurs during a READ that contains no IOSTAT= or END= option, execution of the program terminates.

FORMAT STATEMENTS

Formatted data transfer occurs when a format is given in a data transfer statement. Most often, the format is the statement number of a FORMAT statement. The other possibilities are discussed under the READ statement.

In the following discussion, the term "I/O list" means either an input list or an output list.

FORMAT Statement

label FORMAT (d [,d]...)

label — Mandatory statement label

d — A f ld descriptor or an edit-control descriptor

The parenthesized list of descriptors is known as a format list. Blanks are of no significance in a format list. Parentheses may appear inside a format list to delineate group repeat counts (below).

Field Descriptors: These control the data conversion process during data transfer. For each item in the I/O list, an appropriate field descriptor must be given. Data moving to or from the data item is converted as specified by the corresponding descriptor.

Edit-Control Descriptors: These control more general aspects of the formatting process, such as scale factors, tab control, and the optional printing of literal character items to label the output.

Repeat Counts: A repeat count is an integer constant prefixed to a field descriptor, or to a parenthesized portion or the entirety of a format list. Individual edit-control descriptors can not have repeat counts. As data transfer proceeds, the format list items modified by the repeat count will be re-used the number of times specified before format control proceeds to subsequent format list items. Repeat counts have a maximum nesting of ten levels.

Interaction of the Format and I/O Lists: During data transfer, the format list is scanned from left to right, except as modified by repeat counts. The I/O list is also scanned from left to right.

When an edit-control descriptor is encountered in a format list, the action or alteration required by it is performed. When a field descriptor is encountered, the next I/O list item is edited appropriately and transmitted. If no I/O list items remain when an edit-control descriptor is encountered, data transfer terminates.

When the colon edit-control descriptor is encountered, data transfer terminates if no I/O list items remain to be transmitted; otherwise data transfer continues.

An empty format list may be given to correspond with an empty I/O list.

Rescanning Format Lists: If the format list is exhausted before the I/O list, the file pointer is positioned at the beginning of the next record; format control then reverts to the beginning of the portion of the format list that was terminated by the last preceding right parenthesis. If there is no such parenthesis, format control reverts to the beginning of the format list. Any repeat count preceding the rescanned format is re-used. On output, the current record is padded with blanks and a new record started. On input, the remainder of the current record is skipped, and the file pointer advanced to the beginning of the next record. Reversion of format control, of itself, has no effect on the scale factor, the sign control (S, SP, SS), or the blank control (BN, BZ) in effect at the time of reversion.

Field Descriptors

A field descriptor mediates the conversion of a data item between internal and external form. Usually, the data is supplied by the I/O list. In a character constant field descriptor, it is contained in the descriptor itself.

Numeric Descriptors: I, F, E, D, and G. Unless specified otherwise or modified by edit-control descriptors, the following rules apply to all numeric descriptors:

1. Leading blanks are not significant for input. For output, leading zeroes are suppressed. A minus sign is printed for a negative number, but a positive number is left unsigned.

2. For input with F, E, D, and G descriptors, a decimal point in the input field over-rides the d specification in the descriptor.

3. For output, fields are right justified. If the field width is insufficient, asterisks are produced.

4. Excess digits of precision may be specified on input to non-INTEGER numeric data types. The excess will be ignored.

5. See the BLANK= option of the OPEN statement for the rules concerning blanks in input fields.

The numeric descriptors behave as follows:

- Integer Editing

  Iw[.m]

  Used to edit a short or long integer.

  w is the size of the external field, including blanks and a sign.

  m is the minimum number of places to be displayed on output. Leading zeroes will be printed if necessary to fill the field. For input, m has no effect.

- Real Editing (non-exponential)

  Fw.d

  Writes a real number without an exponent. Reads any real or double precision number.

  w is the size of the field, including blanks, the sign, and the decimal point.

  d is the number of places to the right of the decimal point.

  Input: The decimal point may be omitted from the field. The rightmost d digits will be interpreted as decimal digits. If a decimal point is present, its position over-rides d. Input fields appropriate for E and D editing will also work for F editing.

  Output: d decimal positions are always written.

- Real Editing (Exponential)

  Ew.d[Ee]

  Edits a real or double precision number with an exponent.

  w is the size of the external field, including an exponent and its sign.

  d is the number of decimal places. On input, an explicit decimal point over-rides d.

  e is the number of exponent digits to be displayed on output.

It is ignored for input. When Ee is omitted from an E field descriptor used for output, the defaults listed below under Output will apply.

Input: The exponent may be omitted. E+ØØ will be assumed.

Output: If Ee is present, e digits of the exponent will be printed. If Ee is omitted, the appearance of the exponent will be as follows:

Value of Exponent	Appearance of Exponent
$-99 \leq exp \leq 99$	E ± zz
$-999 \leq exp < -99$	-zzz (no "E")
$99 < exp \leq 999$	+zzz (no "E")
$-9999 \leq exp < -999$	=zzz (fourth digit lost)
$999 < exp \leq 9999$	$zzz (fourth digit lost)

Use of "=" and "$" for overflow exponents is an F77 extension.

Note that the number is always normalized. For non-normalized output, use a scale factor.

- Double Precision Editing

  Dw.d

  Edits a double precision number.

  Input: Operates exactly like an E descriptor.

  Output: Operates exactly like an E descriptor with no Ee present, except that a "D" is substituted wherever an "E" would appear in the output field. For explicit control of double precision exponent format, output the number with an Ew.dEe descriptor.

- Complex Editing

  A complex number consists of a pair of real or double precision numbers. It is edited with an appropriate pair of real or double precision field descriptors. The fact that the two numbers form one entity mathematically is irrelevant to input/output. Edit-control descriptors may appear between the two field descriptors.

- General Editing

  Gw.d[Ee]

  w, d, and e are as defined for the F descriptor.

  Edits real data where the magnitude of the data is not known

beforehand. Produces the more readable F format when possible, but converts to E format when the magnitude of the number exceeds F format representational limits.

Input: The G descriptor is equivalent to the F descriptor.

Output: The G descriptor acts as follows:

Magnitude (M) of Real Data Item	G descriptor acts as:
Ø.1 <= M < 1	F(w-n).d, n('b')
1 <= M < 1Ø	F(w-n).(d-1), n('b')
1Ø <= M < 1ØØ	F(w-n).(d-2), n('b')
.	.
.	.
.	.
1Ø**(d-2) <= M < 1Ø**(d-1)	F(w-n).1, n('b')
1Ø**(d-1) <= M < 1Ø**(d)	F(w-n).Ø, n('b')
Otherwise	Ew.d[Ee]

where: b is a blank
n is 4 for Gw.d and e+2 for Gw.dEe

If M < .Ø1 or M >= 1Ø**d, then Gw.d is equivalent to kPEw.d, where k is the current scale factor.

For input, the Gw.dEe field descriptor is treated identically to the Gw.d descriptor. For output, the Gw.dEe acts as Fw.dEe if Ø.1 <= M < 1Ø**d, and acts as Ew.dEe otherwise.

Non-Numeric Descriptors: L, A, X, B, and format-list character constants.

- Logical Editing

  Lw

  w is the width of the field.

  Input: A valid input field consists of optional blanks, optionally followed by a decimal point, followed by a T or an F. The T or F may be followed by additional characters in the field: they will be ignored.

  Output: The output field consists of w-1 blanks followed by a T or F, as the value of the internal datum is true or false, respectively.

- Character Editing

  A[w]

  w is the width of the field. It is required for input, but optional for output. In the following, L is the length of the character item being edited.

  Input: If w >= L, the rightmost L characters are taken from the external input field. If w < L, the w characters are left justified in the data item and padded with blanks.

  Output: If w > L, the characters are printed right justified in the field, preceded by blanks as needed. If w <= L, the leftmost w characters are printed. If w is not specified it is assumed to be equal to L.

- Character Constant Editing

  'ccc...c' or nHccc...c

  Each c is any ASCII character (not necessarily a member of the F77 character set).

  n is the number of characters in the character constant.

  Character strings in either of these formats may appear as constants in an output format list. Such a string contains its own data, obviating the need for a corresponding item in the output data list. When the string is encountered during the scan of the format list, the characters it contains are written to the current record. A character constant may not appear in a format list used for input, and may not be modified by an individual repeat count.

Note

FORTRAN 66 permitted data to be read into an H format field, altering the value it would print when the format list involved was later used for output. FORTRAN 77 will not accept this practice.

- Space Skipping

nX

n is an integer. On output, equivalent to a character constant of n blanks. On input, equivalent to TRn. No repeat count may appear.

- Business Editing

B 'string'

The B descriptor is used in printing business reports where it is desirable to fill number fields to prevent unauthorized modifications (as on checks), suppress leading zeroes and plus signs, print trailing minus signs (accounting convention) and convert minus signs to CR (for indicating credit entries on bills). Business editing is an F77 extension.

The length of the string determines the field width. If the width is too small for the number, then the output will be a string of asterisks filling the field. Legal characters for the string are:

+ - $, * Z # . CR

Plus (+):

If only the first character is +, then the sign of the number (+ or -) is printed in the leftmost portion of the field (Fixed sign). If the string begins with more than one + sign, then these will be replaced by asterisks and the sign of the number (+ or -) will be printed in the field position immediately to the left of the first printing character of the number (Floating sign). If the rightmost character of the string is +, then the sign of the number (+ or -) will be printed in that field position following the number (Trailing sign).

Minus (-):

Behaves the same as a plus sign except that a space (blank) is printed instead of a + if the number is positive (Plus sign suppression).

Dollar Sign ($):

A dollar sign ($) may at most be preceded in the string by an optional fixed sign. A single dollar sign will cause a $ to be printed in the corresponding position in the output field (Fixed dollar).

Multiple dollar signs will be replaced by printing characters in the number and a single $ will be printed in the position immediately to the left of the leftmost printing character of the number (Floating dollar).

Asterisk (*):

Asterisks may be preceded only by an optional fixed sign and/or a fixed dollar. Asterisks in positions used by digits of the number will be replaced by those digits; the remainder will be printed as asterisks (Field filling).

Zed (Z):

If the digit corresponding to a Z in the output number is a leading zero, a space (blank) will be printed in that position; otherwise the digit in the number will be printed (Leading-zero suppression).

Number sign (#):

#'s indicate digit positions not subject to leading-zero suppression; the digit in the number will be printed in its corresponding portion whether zero or not (Zero non-suppression).

Decimal point (.):

Indicates the position of the decimal point in the output number. Only #'s and either trailing signs or credit (CR) may follow the decimal point.

Comma (,):

Commas may be placed after any leading character, but before the decimal points. If a significant character of the number (not a sign or dollar) precedes the comma, a , will be printed in that position. If not preceded by a significant character, a space will be printed in this position unless the comma is in an asterisk field; then an * will be printed in that position.

Credit (CR):

The characters CR may only be used as the last two (rightmost) of the string. If the number is positive, 2 spaces will be printed following it; if negative, the

letters CR will be printed.

See Table 4-4 for examples of B-Format usage.

Edit-Control Descriptors

These control general aspects of the formatting process. They differ from field descriptors in that they do not correspond to or supply individual data items, but modify the environment in which the data transfer process occurs.

- Scale Factors

  kP

  where scale factor k is an unsigned or negative integer constant. The comma following a scale factor is often omitted, so that it becomes a prefix of a subsequent field descriptor. The scale factor has various effects, depending on the descriptor type and the direction of data transfer.

  F, E, D, and G Input: If there is an exponent in the field, the scale factor has no effect. Otherwise, it converts the data so that:

  External Value = Internal Value*(1Ø**k)

  F Output: The scale factor converts the value as for F input.

  E and D Output: The mantissa is multiplied by 1Ø**k and the exponent is reduced by k to maintain the same overall value. This permits output of E and D numbers in non-normalized form.

  G Output: If the G is acting as an F, the scale factor is ignored. If it is acting as an E, the scale factor behaves as described for E output.

  Note

  Once a scale factor has been used, it remains in effect for all subsequent descriptors of appropriate type, until it is reset to another value or to zero. When a format list is rescanned, the scale factor is not reset to zero automatically. If a scale factor is to affect only one field, "ØP" must appear before the next scalable descriptor that occurs.

Table 4-4. Examples of B-Format Usage

Number	Format	Output Field
123	B'####'	Ø123
12345	B'####'	****
Ø	B'####'	ØØØØ
123	B'ZZZZ'	123
1234	B'ZZZZ'	1234
Ø	B'ZZZZ'	
Ø	B'ZZZ#'	Ø
1.Ø35	B'#.##'	1.04
Ø	B'#.##'	Ø.ØØ
1234.56	B'ZZZ,ZZZ,ZZ#.##'	1,234.56
123456.78	B'ZZZ,ZZZ,ZZ#.##'	123,456.78
Ø	B'ZZZ,ZZZ,ZZ#.##'	Ø.ØØ
2	B'+###'	+ØØ2
-2	B'+###'	-ØØ2
2	B'-ZZ#'	2
-2	B'-ZZ#'	- 2
234	B'ZZZZZ+'	234+
-234	B'ZZZZZ+'	234-
234	B'ZZZZZ-'	234
-234	B'ZZZZZ-'	234-
12345	B'ZZZ,ZZ#CR'	12,345
-12345	B'ZZZ.ZZ#CR'	12,345CR
123	B'+++,++#.##'	123.ØØ
-123	B+++,++#.##'	-123.ØØ
98	B'$ZZZZZZ#'	$ 98
98	B'$$$$$$$#'	$98
156789	B'$***,***,**#.##'	$****156,789.ØØ

- Sign Control Editing

  SP SS S

  These control the placement of plus signs in numeric output. Once a sign control descriptor is encountered, it remains in effect until it is explicitly altered or revoked.

  SP: The processor will insert a plus sign wherever one may optionally appear.

  SS: The processor will not insert any plus sign whose appearance is optional.

  S: The processor will return to the locally defined system default for sign editing.

- Blank Control Editing

  BN BZ

  The method of handling blanks in numeric input fields that is established for a file by the BLANK= option of the OPEN statement may be temporarily over-ridden by BN or BZ. The method may be altered as often as desired, and will revert to the BLANK= value when the READ statement is complete. Blank control descriptors have no effect on output.

  BN: All blanks will be deleted, and digits compressed to the right side of the input field. An all-blank field is interpreted as a zero value.

  BZ: All but leading blanks will be converted to zeroes, as in FORTRAN 66.

- Positional Editing

  Tn TLn TRn

  where n is an integer constant.

  The following description presupposes that you have read about the I/O buffer F$IOBF at the beginning of the subsection on DATA TRANSFER STATEMENTS, above. The pointer which scans F$IOBF during data transfer ordinarily behaves as follows:

  1. Before data transfer, it points to the first position (byte) of F$IOBF.

  2. While an F$IOBF position is being read or written, it points to that position.

3. After a position has been read or written, it moves to the next position to the left and remains there.

4. After the last F$IOBF position has been read or written, it remains at that position.

Note that this behavior is the same as that of the carriage position on an ordinary typewriter.

The T edit-control descriptor is used to alter the sequential progress of the F$IOBF pointer. The pointer can be moved to the left or right of its current position, or to an absolute position, in any desired sequence. Subsequent data transfers will begin at the new position. Thus F$IOBF positions, and hence the corresponding current-record positions, can be accessed as often as desired and in any order.

If an attempt is made to move the F$IOBF pointer beyond the first (or last) F$IOBF position, the pointer will stop and remain at that position. If T descriptors are used during a WRITE in such a way that some F$IOBF positions remain undefined after all data items have been transferred, the undefined positions will be filled with blanks before F$IOBF is written to the current file record.

Moving the F$IOBF pointer has no effect on the file pointer, which never skips positions within a record. Beware of confusing these two pointers.

TLn: Move the F$IOBF pointer n positions left.

TRn: Move the F$IOBF pointer n positions right.

Tn: Move the F$IOBF pointer to the nth character of the record.

- Conditional Output

:

A colon placed in a format list will cause data transfer to terminate at that point if no items remain in the output list. This feature is most often used to increase the versatility of a format list which contains character constant descriptors used in labeling the output. A colon is ignored on input.

- Record Skipping

/[/]...

A slash in a format list causes I/O processing to proceed to the next record. As many new records will be begun as there are slashes. The effect of slashes at the beginning or end of a format list is additional to the automatic beginning of a new

record with each data transfer statement.

Input: Under sequential access, a slash causes the remaining portion of the current record to be skipped, and the file pointer to be positioned at the beginning of the next record, making it the current record. Under direct access, the remainder of the record is skipped, the record number increased by one, and the file pointer positioned at the beginning of the record that has that record number.

Output: Similar to input, except that all positions skipped over will be filled with blanks.

Commas adjacent to slashes may be omitted.

SUMMARY OF STATEMENT SYNTAX

In the following table, all input/output statements are listed in alphabetical order with their syntax requirements. The table is intended only as a reminder for those already familiar with the statements.

Table 4-5.

Input/Output Statement Syntax

Statement	Syntax
BACKSPACE	BACKSPACE ([UNIT=]unit# [,IOSTAT=ios] [,ERR=label])
CLOSE	CLOSE ([UNIT=]unit# [,STATUS= stat] [,ERR= label] [,IOSTAT= ios])
ENDFILE	ENDFILE ([UNIT=]unit# [,IOSTAT= ios] [,ERR= label])
FORMAT	label FORMAT (d [,d]...)
INQUIRE	INQUIRE ([FILE=]filename or [UNIT=]unit# [,ERR= s] [,EXIST= ex] [,OPENED= od] [,NUMBER= num] [,NAMED= nmd] [,NAME= fn] [,ACCESS= acc] [,SEQUENTIAL= seq] [,DIRECT= dir] [,FORM= fm] [,FORMATTED= fmt] [,UNFORMATTED= unf] [,RECL= rcl] [,NEXTREC= nr] [,BLANK= blnk] [,IOSTAT= ios])
OPEN	OPEN ([UNIT=]unit# [,FILE= filename] [,STATUS=stat] [,ACCESS= acc] [,FORM= fm] [,RECL= reclength] [,BLANK= blnk] [,ERR= label] [,IOSTAT= ios]
PRINT	PRINT format [,output list]
READ Sequential:	READ ([UNIT=]unit# [,[FMT=]format] [,END= label] [,ERR= label] [,IOSTAT= ios]) [input list]
READ direct ANS:	READ ([UNIT=]unit# [,[FMT=]format] ,REC= record# [,END=label] [,ERR=label] [,IOSTAT=ios])[input list]
READ direct IBM:	READ (unit#'record# [,[FMT=]format] [,END= label] [,ERR= label] [,IOSTAT= ios]) [input list]
REWIND	REWIND ([UNIT=]unit# [,IOSTAT= ios] [,ERR= label])
WRITE Sequential:	WRITE ([UNIT=]unit# [,[FMT=]format] [,ERR= label] [,IOSTAT= ios]) [output list]
WRITE direct ANS:	WRITE ([UNIT=]unit# [,[FMT=]format] ,REC= record# [,ERR= label] [,IOSTAT= ios]) [output list]
WRITE direct IBM:	WRITE (unit#'record# [,[FMT=]format] [,ERR= label] [,IOSTAT= ios]) [output list]

SECTION 5

SUBROUTINES AND FUNCTIONS

In addition to one main program, a FORTRAN 77 program may contain any number of subroutines and functions, collectively called subprograms. These may be drawn from existing libraries, or may be supplied by the programmer.

Special terms used below are defined at the beginning of Section 2.

SUBROUTINES

All subroutines are referenced in the same way, regardless of origin:

```
CALL name [( [argument [,argument]...] )]
```

where name is the name of the subroutine, and the arguments are a list of data items agreeing in number, order, and type with the dummy argument list in the subroutine's header statement. If the argument list is empty, the parentheses may be omitted. Constants and expressions are permissible as arguments. Any character expressions must be fixed-length.

When control reaches a subroutine CALL, the subroutine executes. Data is returned by alteration of the values of arguments and of data in COMMON. Data must not be returned to an actual argument that was an expression: no error message will be printed but invalid results may occur.

Caution

In FORTRAN 77, arguments are passed by reference (address). Therefore it is extremely important not to alter the value of a dummy argument whose actual argument is a constant or a parameter (a constant item). Such an alteration will alter the value kept in storage for the constant item, just as it would for a variable. If the compiler has utilized the same storage copy of the constant item in coding other references to the item, the altered value will be used when the code is executed. Example:

```
      I = 5
      PRINT 1Ø,I                  /* Value printed is 5
      CALL SUB1(5)
      I = 5
      PRINT 1Ø,I                  /* value printed is 25.
   1Ø FORMAT (I2)
      STOP
      END

      SUBROUTINE SUB1 (J)
      J = J**2
      RETURN
      END
```

Subroutine Libraries

Prime supplies several libraries of subroutines. These allow PRIMOS subroutines to be called from within an F77 program, and provide various commonly used utilities. When a subroutine from such a library has been called from a program, the command:

LI library-name

must be given to SEG at load time before the unqualified LI command is given.

For more information, see The PRIMOS Subroutines Reference Guide.

Note

Many PRIMOS subroutines require and return short integer arguments. When long integers are used to supply data to such a subroutine, convert them directly in the argument list with the INTS intrinsic function. Arguments to which data is returned must themselves be short integers, since data cannot be returned to an expression.

User-Supplied Subroutines

These are constructed as follows:

```
SUBROUTINE name [( [argument [,argument]...] )]

(any number of FORTRAN 77 statements)

[RETURN]

END
```

The name is any legal F77 name having not more than eight characters. The arguments are a list of dummy arguments corresponding to actual arguments passed by the calling program unit. A dummy argument may be:

- A variable name
- An array name
- A dummy subprogram name
- An asterisk

Variable and array names must be typed and dimensioned as with any other such names. The actual argument corresponding to a dummy subprogram name must have been declared INTRINSIC or EXTERNAL in the calling program unit. An asterisk corresponds to an alternate return specifier.

There is no syntactic upper limit to the number of arguments.

Alternate Returns: A subroutine can return to the statement following the point of call - this is the usual action - or it can return to any labeled executable statement in the calling program unit. The subroutine can select the statement to which it will return. Alternate returns are accomplished as follows:

1. The label of every statement to which the subroutine might return must appear in the argument list of the CALL statement, prefixed by an asterisk (or a dollar sign - F77 extension).

2. An asterisk appears in the dummy argument list of the subroutine at each position corresponding to a statement label in the CALL statement.

3. RETURN statements in the subroutine may optionally be followed by an integer expression n. When control encounters a RETURN n in the subroutine, the subroutine will return to the statement of the calling program unit whose label corresponds to the n'th asterisk in the dummy argument list of the subroutine. If control first encounters a RETURN without a number, or with a number outside the applicable range, a return to the point of call will occur. Example:

```
      PROGRAM ALTRTRN
100   CALL PROC1 (J)
300   CALL PROC2 (K)
500   CALL PROC3 (J, K, *100, 4, *900)
700   GO TO 100
900   STOP
      END

      SUBROUTINE PROC3 (J,K,*,M,*)
      IF (I .EQ. J) RETURN
      IF ((I + J) .EQ. K) RETURN 1      /* Returns to stmt 100
      IF ((I + K) .EQ. J) RETURN M/2    /* Returns to stmt 900
      RETURN                            /* Returns to stmt 700
      END
```

Alternate returns are permitted following the CALL of a subroutine at a secondary entry point (see below). Only the asterisks in the dummy argument list at the point of entry are counted.

Recursion

In FORTRAN 77, recursion is not permitted. F77 has been extended to permit recursion in subroutines, though not in functions. The rules and syntax are identical in recursive and non-recursive subroutine calls.

FUNCTIONS

All functions are referenced in the same way, regardless of origin:

```
[target =] name ( [argument [,argument]...] )
```

where name is the name of the function, and the arguments are a list of data items defined as for subroutines. Note that the parentheses are mandatory, even when the argument list is empty.

When control reaches a function reference, the value is calculated and made available at the point of the reference. If the reference is part of an assignment statement, type conversion will occur if appropriate, as with any other assignment. A function may return additional data by altering the values of its arguments and of data in COMMON.

FORTRAN 77 functions cannot be referenced recursively.

Caution

When a function reference appears in an expression, evaluation of the function must not alter the value of any other entity in the expression, either directly or by altering arguments to other functions.

Intrinsic Functions

F77 supplies numerous built-in functions, known as intrinsic functions, which may be invoked at any point in an F77 program unit. These are discussed in Section 6.

User-Supplied Functions

These are constructed as follows:

```
[type] FUNCTION name ( [argument [,argument]...] )

(any number of FORTRAN 77 statements)

[RETURN]

END
```

where type is any F77 data type, and name and the arguments are defined as for subroutines. Note that the parentheses are required even if there are no arguments. There may be any number of RETURNs but alternate returns are not permitted; hence no asterisks may appear in the argument list. END implies RETURN automatically.

The name of the function must appear as a variable within the function. It may be used like any other variable, and must be defined by the time the function returns. Its value at that time becomes the value of the function.

Statement Functions

Any function that can be expressed as a single statement may appear within the body of a program unit. It may be referenced only from within that unit. The form is:

```
name ( [argument] [,argument]...] ) = expression
```

where name and the dummy arguments are defined as for any function, and are typed in any standard way.

A dummy argument name in a statement function is defined only within that function; if it is duplicated in another statement function, the two names have no connection.

If a dummy argument name in a statement function duplicates the name of a data item in the containing program unit, that name will refer to the actual argument associated with it in the function reference, rather than to the similarly named program-unit data item.

The expression may make use of any data item available within the

program unit containing the statement function, except one whose name duplicates that of a dummy argument and does not appear in the actual argument list at the function reference.

A statement function may reference any previously defined statement function, but may not reference itself. Statement functions are local to the program unit in which they are defined: they may not be passed as arguments.

SECONDARY ENTRY POINTS

A subset of a subprogram can be executed as if it were a separate program unit, and given its own argument list, by using the ENTRY statement. Consecutive ENTRY statements can appear, so that execution may begin at a given point with any of a variety of argument lists. The form of the ENTRY statement is:

```
ENTRY name [( [argument [,argument]...] )]
```

where name and the arguments are defined as for a subroutine. The argument list of an ENTRY statement need not correspond with that in the header or any other ENTRY statement.

A secondary entry is referenced (in a function) or CALLed (in a subroutine) exactly as the main entry point would be, and supplied arguments corresponding to its particular argument list. Control proceeds from the entry point to the first RETURN or END encountered. ENTRY statements encountered in-line are ignored.

An entry name to a function may be typed by default or in a type-statement. The type may differ from that of the function name and of other entry names, except that all entry names in a CHARACTER function must be of type CHARACTER and have the same *(length) specification. All entry names in a function are automatically equivalenced. Before the function returns, assignment to an entry name of the same type as the entry name used in referencing the function must occur.

Alternate returns are permitted following the CALL of a subroutine at an entry point. Only the statement labels in the entry point's argument list are counted.

Note

In some versions of FORTRAN IV, the association of actual and dummy arguments established when a subprogram is invoked at any entry point persists following return to the invoking program unit. Consequently, a subprogram can be invoked repeatedly at various entry points, and reference made after each invocation to any dummy argument that became associated with an actual argument at any previous invocation. This technique is not accepted by any Prime FORTRAN.

ADJUSTABLE SUBPROGRAM ELEMENTS

The length of the value returned by a type CHARACTER function, the lengths of type CHARACTER dummy arguments in a subprogram, and the dimension bounds of an array dummy argument, can be made adjustable. An adjustable element will take on the length or bounds of the corresponding actual argument at each call. Such flexibility can considerably increase the versatility of a subprogram.

Adjustable Character Functions

To make a CHARACTER function adjust the length of its result, specify its length as an asterisk in parentheses:

```
CHARACTER*(*) FUNCTION CFUNC (A,B)
```

In each program unit referencing the adjustable function, use a type-statement to assign the CHARACTER type and a length to the name of the function. The length of the value returned at each function reference will be the one assigned to the function in the referencing program unit.

Adjustable Character Arguments

To make a type CHARACTER dummy argument adjustable, specify its length to be (*) in a type-statement:

```
SUBROUTINE YORD (CVAR)
CHARACTER*(*) CVAR
```

CVAR will take on the length of the actual argument corresponding to it at each call.

Assumed-Size Arrays

To create an assumed-size array, replace the upper bound of the last dimension specification in a fixed or adjustable dummy array declaration with an asterisk. That dimension will take on the upper bound associated with it in the corresponding actual array in the calling program unit.

Adjustable Array Dimensions

To create an adjustable array, pass the name of an existing array to an appropriately typed dummy argument in a subprogram. Dimension the dummy array using:

1. Integer variables passed to integer dummy arguments in the subprogram, and/or

2. Integer variables from COMMON.

Expressions are permitted in adjustable array bound declarations, subject to the following restrictions:

- All variables must be INTEGER
- No array references
- No function references

Example:

```
REAL FUNCTION ARRTEST(ANAME, DIM1, DIM2)
IMPLICIT INTEGER (A-Z)
COMMON /BND/ DIM3,N
DIMENSION ANAME (DIM1, DIM2:N, 1:1Ø, DIM3+12)
```

When control passes to a subprogram containing an adjustable array, the array bounds are determined before execution begins. The variables used may therefore be redefined or become undefined during execution without affecting the dimensional properties of the array.

Caution

Adjustable arrays do not represent dimension-by-dimension subsets of the original array, but are equivalenced to the original array as a whole. The adjustable array cannot be longer than the corresponding actual array.

ARRAYS AS ARGUMENTS

The F77 compiler can produce two types of object code. Ordinary code can address only within a segment. Boundary-spanning code is capable of addressing across the boundary between one segment and the next.

Whenever an array extends across a segment boundary, all references to it must consist of boundary-spanning code, because those portions of it in segments higher than the one in which it begins are inaccessible to ordinary code.

Arrays in local static or dynamic storage present no problem, because there may be at most one segment for all static variables, and another for all dynamic variables: hence no boundary-spanning is possible. Arrays in COMMON blocks under 128K bytes (one segment) long present no problem because such blocks are always loaded within a single segment.

An array in a COMMON block over one segment long (a large COMMON block) may or may not span a segment boundary, depending on its size and its location in the block. In practice, no array under one segment long should ever be placed in a large COMMON block - see the Note below.

When a program unit is compiled, the F77 compiler inspects any COMMON statements in it for COMMON block size and the presence of arrays. All references in the program unit to any array the compiler knows to be in a large COMMON block will automatically be compiled with boundary-spanning code. No special action is required of the programmer in this case.

However, when a dummy array occurs in a subprogram, the compiler has no way to know the storage status of any actual array that will become associated with it when the subprogram is invoked. Therefore, the compiler cannot know whether to compile references to the dummy array with ordinary or boundary-spanning code. It is the programmer's responsibility to inform the compiler of the correct action in this case, through use of of the -BIG/-NOBIG compiler option.

When a subprogram is compiled with -NOBIG (the default), dummy array references within it will be compiled with ordinary code; the actual array passed to any dummy array in it must then be contained within one segment. When a subprogram is compiled with -BIG, all references it makes to any dummy array will be compiled with boundary-spanning code; the actual array passed to any dummy argument in it may then span a segment boundary, though it need not do so.

A dummy-array reference compiled with boundary-spanning code will execute correctly for any actual array, whether it spans a segment boundary or not. However, boundary-spanning code executes more slowly than ordinary code because it performs more complex address calculation. The -BIG option should therefore not be used unnecessarily.

Note

> An array less than 128K bytes long should not be put in a large COMMON block, since this will cause the inefficiency of boundary-spanning code to be needlessly incurred in every reference to the array.

Character Arrays as Arguments

When a CHARACTER array that may cross a segment boundary is passed as an argument, the element size of the actual array and the dummy array must be the same. This is an F77 restriction required to insure that no element of the array can fall across a segment boundary. See the COMMON Statement in Section 3 for more information on COMMON block restrictions.

SUBPROGRAMS AS ARGUMENTS

Entire subprograms may be passed as arguments to other subprograms, where they may be referenced or passed again. The general method is as follows:

1. In the invoking program unit, name any intrinsic functions to be passed in an INTRINSIC statement, and any user-supplied or library subprograms to be passed in an EXTERNAL statement.

2. In the actual argument list for each invocation, name the subprograms which are to be passed to the invoked subprogram.

3. In the dummy argument list at the entry point to the invoked subprogram (either its header or an ENTRY statement), place an untyped dummy subroutine name at each position corresponding to an actual argument that is a subroutine, and an appropriately typed dummy function name at each position corresponding to an actual argument that is a function.

4. In the invoked subprogram, use the appropriate dummy subprogram name wherever a reference to the corresponding actual subprogram is desired.

For example, suppose that MAIN wishes to call SUB repeatedly, passing at each call one of the intrinsic functions DSIN and DCOS, and one of the user-supplied subroutines GREATER and LESSER. The code could be as follows:

```
PROGRAM MAIN
INTRINSIC  DSIN, DCOS
EXTERNAL  GREATER, LESSER
CALL SUB (DSIN, GREATER, 1.DØ)
CALL SUB (DCOS, GREATER, 1.DØ)
CALL SUB (DCOS, LESSER, 1.DØ)
STOP
END

SUBROUTINE SUB (TRIG, COMPARE, NUM)
DOUBLE PRECISION TRIG, NUM
IF (TRIG(NUM) .GT. DTAN (NUM)) CALL COMPARE (NUM)
RETURN
END
```

Not all intrinsic functions can be passed as arguments. See Section 6 before passing intrinsic functions.

SECTION 6

INTRINSIC FUNCTIONS

For general information on the use of functions in F77 programs, see Section 5.

F77 INTRINSIC FUNCTIONS

FORTRAN 77 supplies a wide variety of intrinsic (built-in) functions. These are used for type conversion, character data evaluation, lexical comparison, and the calculation of various mathematical quantities.

The F77 intrinsic function set includes all FORTRAN 77 intrinsics, plus additional functions for bitwise logical operations, bitwise shifts, truncation of an integer, determination of a data item's storage address, and operations on the COMPLEX*16 data type.

All F77 intrinsic functions are built in to the language. They may be invoked at any point in any F77 program unit. The F77 compiler and the SEG linking loader will automatically supply the functions invoked: no additional action by the programmer is required.

It is recommended that the names of any F77 intrinsic functions invoked in a program unit appear in an INTRINSIC statement in that unit. This practice will result in immediate diagnostic messages if the program is run on a different system which does not supply all the needed intrinsics.

Generic and Specific Functions

Many FORTRAN 77 intrinsic functions are generic: they exist in several versions, called specific functions, which differ only in the data type each accepts. When the programmer references a generic function, the F77 compiler will examine the argument list at the reference and select the specific function appropriate to the data type of the arguments.

Not all specific functions are individually named. Those that are may be invoked directly by name, in which case the programmer must be careful to supply the correct data types.

Intrinsic Functions as Arguments

See SUBPROGRAMS AS ARGUMENTS in Section 5. The following is additional to the discussion there.

Only named specific functions can be passed as arguments to subprograms. In some cases, a specific function has the same name as its generic function. When this name appears in an argument list, it is the specific function that is passed.

Intrinsic functions for type conversion, selection of a maximum or minimum value, lexical comparison, logical operation, shifting, truncation of bits, and determination of a data item's memory address cannot be passed as arguments.

Long and Short Integer Arguments

All new programs written in F77 should use long integers exclusively, in conformance with the ANSI standard. When program units are converted from FORTRAN IV to F77, or when F77 program units are written which will return values to an existing FORTRAN IV program unit, the use of short integers in an F77 program unit may become necessary.

No constraint on the use of short integers is imposed by the F77 intrinsic set. All F77 intrinsic functions have been extended to accept either long or short arguments, or a mixture of the two, and to produce short integer results where appropriate. ANS FORTRAN 77 does not provide short integers or permit data types to be mixed in an intrinsic function's argument list.

An intrinsic function which produces an integer result (an integer intrinsic) will produce either a long or short integer. For integer intrinsics other than INT whose arguments are integers, the result-type depends on the argument list at the particular invocation. For integer intrinsics whose arguments are not integers, and for INT, the result-type depends on the compiler option (-INTS or -INTL) in effect when the program unit containing the intrinsic was compiled. The notes for Table 6-1 tell more exactly how the result-type for each integer intrinsic is determined.

TABLE OF INTRINSIC FUNCTIONS

The following table, with its accompanying notes, provides a complete description of the F77 intrinsic functions. Where a specific F77 function has the same name as an existing FTN function, the functions are the same, except as noted under Reimplemented FTN Constructs in Appendix A. Before using any function with which you are not completely familiar, be sure to study carefully the table entry and accompanying notes, if any, for that function.

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Type Conversion	Numeric to Integer	1	INT	-	Integer	Integer	1,32,34,35
				INT	Real	Integer	
				IFIX	Real	Integer	
				IDINT	Double	Integer	
				-	Complex	Integer	
				-	Complex*16	Integer	
	Numeric to Short Integer	1	INTS	-	Integer	Integer*2	2,34,36
				-	Real	Integer*2	
				-	Double	Integer*2	
				-	Complex	Integer*2	
				-	Complex*16	Integer*2	
	Numeric to Long Integer	1	INTL	-	Integer	Integer*4	2,34,36
				-	Real	Integer*4	
				-	Double	Integer*4	
				-	Complex	Integer*4	
				-	Complex*16	Integer*4	
	Numeric to Real	1	REAL	FLOAT	Integer	Real	3,34,35
				-	Real	Real	
				SNGL	Double	Real	
				-	Complex	Real	
				REAL	Complex*16	Real	
	Numeric to Double Precision	1	DBLE	-	Integer	Double	4,34,35
				-	Real	Double	
				-	Double	Double	
				-	Complex	Double	
				DREAL	Complex*16	Double	
	Numeric to Complex	1 or 2	CMPLX	-	Integer	Complex	5,34,35
				-	Real	Complex	
				-	Double	Complex	
				-	Complex	Complex	
				-	Complex*16	Complex	

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
	Numeric to Complex*16	1 or 2	DCMPLX	-	Integer	Complex*16	6,34,36
				-	Real	Complex*16	
				-	Double	Complex*16	
				-	Complex	Complex*16	
				-	Complex*16	Complex*16	
	Character to Integer	1	-	ICHAR	Character	Integer	7,32
	Integer to Character	1	-	CHAR	Integer	Character	7
Truncation	REAL(INTL(a))	1	AINT	AINT	Real	Real	
	DBLE(INTL(a))			DINT	Double	Double	
Nearest Whole Number	See Note 8	1	ANINT	ANINT	Real	Real	8
				DNINT	Double	Double	
Nearest Integer	See Note 9	1	NINT	NINT	Real	Integer	9,32
				IDNINT	Double	Integer	
Absolute Value	(a**2)**.5	1	ABS	IABS	Integer	Integer	10,16,35
				ABS	Real	Real	
				DABS	Double	Double	
	(ar**2+ai**2)**.5			CABS	Complex	Real	
				CDABS	Complex*16	Double	
Remaindering	See Note 11	2	MOD	MOD	Integer	Integer	11,33
				AMOD	Real	Real	
				DMOD	Double	Double	
Transfer of Sign	|a1| if a2 >= 0	2	SIGN	ISIGN	Integer	Integer	12,33
	-|a1| if a2 < 0			SIGN	Real	Real	
				DSIGN	Double	Double	
Positive Difference	a1-a2 if a1 > a2	2	DIM	IDIM	Integer	Integer	33
	0 if a1 <= a2			DIM	Real	Real	
				DDIM	Double	Double	

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Double Precision Product	a1*a2	2	-	DPROD	Real	Double	
Choosing Largest Value	max(a1,a2,...)	>= 2	MAX	MAXØ AMAX1 DMAX1	Integer Real Double	Integer Real Double	33,34
			-	AMAXØ MAX1	Integer Real	Real Integer	32,34
Choosing Smallest Value	min(a1,a2,...)	>= 2	MIN	MINØ AMIN1 DMIN1	Integer Real Double	Integer Real Double	33,34
			-	AMINØ MIN1	Integer Real	Real Integer	32,34
Length	Length of Character Entity	1	-	LEN	Character	Integer	13,32
Index of a Substring	Location of Substring a2 in String a1	2	-	INDEX	Character	Integer	14,32
Real Part of Complex Argument	ar	1	-	REAL DREAL	Complex Complex*16	Real Double	15,16,
Imaginary Part of Complex Argument	ai	1	-	AIMAG DIMAG	Complex Complex*16	Real Double	16,35
Conjugate of a Complex Argument	(ar,-ai)	1	CONJG	CONJG DCONJG	Complex Complex*16	Complex Complex*16	16,35
Square Root	(a)**.5	1	SQRT	SQRT DSQRT CSQRT CDSQRT	Real Double Complex Complex*16	Real Double Complex Complex*16	17,35

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Exponential	e**a	1	EXP	EXP DEXP CEXP CDEXP	Real Double Complex Complex*16	Real Double Complex Complex*16	35
Natural Logarithm	log(a)	1	LOG	ALOG DLOG CLOG CDLOG	Real Double Complex Complex*16	Real Double Complex Complex*16	18,35
Common Logarithm	log10(a)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double	18
Sine	sin(a)	1	SIN	SIN DSIN CSIN CDSIN	Real Double Complex Complex*16	Real Double Complex Complex*16	19,21,35
Cosine	cos(a)	1	COS	COS DCOS CCOS CDCOS	Real Double Complex Complex*16	Real Double Complex Complex*16	19,21,35
Tangent	tan(a)	1	TAN	TAN DTAN	Real Double	Real Double	19,21
Arcsine	arcsin(a)	1	ASIN	ASIN DASIN	Real Double	Real Double	20,22
Arccosine	arccos(a)	1	ACOS	ACOS DACOS	Real Double	Real Double	20,23
Arctangent	arctan(a)	1	ATAN	ATAN DATAN	Real Double	Real Double	20,24
	arctan(a1/a2)	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double	20,24

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Hyperbolic Sine	sinh(a)	1	SINH	SINH DSINH	Real Double	Real Double	19
Hyperbolic Cosine	cosh(a)	1	COSH	COSH DCOSH	Real Double	Real Double	19
Hyperbolic Tangent	tanh(a)	1	TANH	TANH DTANH	Real Double	Real Double	19
Lexically Greater Than or Equal	a1 >= a2	2	-	LGE	Character	Logical	25,34
Lexically Greater Than	a1 > a2	2	-	LGT	Character	Logical	25,34
Lexically Less Than or Equal	a1 <= a2	2	-	LLE	Character	Logical	25,34
Lexically Less Than	a1 < a2	2	-	LLT	Character	Logical	25,34
Logical Operations	Bitwise AND	Any	-	AND	Integer	Integer	26,34,36
	Bitwise OR	Any	-	OR	Integer	Integer	26,34,36
	Bitwise XOR	Any	-	XOR	Integer	Integer	26,34,36
	Bitwise NOT	1	-	NOT	Integer	Integer	27,34,36
Shifts	Shift Left	2	-	LS	Integer	Integer	28,34,36
	Shift Right	2	-	RS	Integer	Integer	28,34,36
	Shift	2 or 3	-	SHFT	Integer	Integer	29,34,36

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Truncation	Truncate Left	2	-	LT	Integer	Integer	30,34,36
	Truncate Right	2	-	RT	Integer	Integer	30,34,36
Storage Address	Actual Storage Address of Data Item	1	-	LOC	See Note 31	Integer*4	31,34,36

NOTES FOR THE TABLE OF INTRINSIC FUNCTIONS

In the following notes the names of data types are given in lowercase; uppercase is reserved for intrinsic function names.

1 The generic INT discards the fractional part of its argument, producing a truncated (unrounded) integral value. The result will be INTEGER*2 in a program unit compiled with -INTS, and INTEGER*4 in a program unit compiled with -INTL (the default).

2 INTS and INTL are similar to INT, differing only in that the result-type is determined by the function selected rather than the compiler option in effect.

3 For a of type real, REAL(a) is a. For a of type integer or double precision, REAL(a) is as much precision of a as a real datum can contain. For a of type complex, REAL(a) is the real part of a.

4 For a of type double precision, DBLE(a) is a. For a of type integer or real, DBLE(a) is the value of a in double precision form. For a of type complex, DBLE(a) is the real part of a in double precision form.

5 CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or complex. If there are two arguments, they must both be of the same type and may be of type integer, real, or double precision.

For a of type complex, CMPLX(a) is a. For a of type integer, real, or double precision, CMPLX(a) is the complex value whose real part is REAL(a) and whose imaginary part is zero.

CMPLX(a1,a2) is the complex value whose real part is REAL(a1) and whose imaginary part is REAL (a2).

6 DCMPLX is similar to CMPLX, except that a complex*16 number is produced.

7 Every ASCII character is represented in the computer as a sequence of eight bits ranging from 10000000 to 11111111 (octal :200 to :377, Decimal 128 to 255). Any such sequence can be interpreted either as an ASCII character or as an integer. CHAR and ICHAR provide a means for converting between the two interpretations.

ICHAR operates on a single ASCII character. It returns an integer between 128 and 255, representing the decimal equivalent of the ASCII bit pattern for that character.

CHAR operates on any integer. If the integer is between 128 and 255, it is used directly. If the integer is not between 128 and 255, it is converted to one that is, as follows:

1. Truncate all but the eight rightmost bits (the lowest-order byte).

2. Set the leftmost remaining bit to 1.

Following conversion if required, CHAR returns the ASCII character whose bit pattern corresponds to the binary equivalent of its argument.

The effect of the conversion is that for every integer I

```
CHAR(I) = CHAR(MOD(I,128) + 128)
```

The ASCII character set is described in Appendix D.

8 ANINT(a) is defined as:

```
REAL(INTL(a+.5)) if a >= Ø
REAL(INTL(a-.5)) if a < Ø
```

DNINT(a) is defined as:

```
DBLE(INTL(a+.5)) if a >= Ø
DBLE(INTL(a-.5)) if a < Ø
```

9 NINT(a) and IDNINT(a) are defined as:

```
INT(a+.5) if a >= Ø
INT(a-.5) if a < Ø
```

10 The argument to IABS may be INTEGER*2 or INTEGER*4. The result will be of the same type as the argument.

11 MOD yields the remainder when its first argument is divided by its second argument. Both arguments must be of the same type; the result will also be of that type.

The three specific functions under MOD are defined:

```
MOD(a1,a2) = a1 - (INTL(a1/a2) * a2)
AMOD(a1,a2) = REAL(a1 - (INTL(a1/a2) * a2))
DMOD(a1,a2) = DBLE(a1 - (INTL(a1/a2) * a2))
```

The result for MOD, AMOD, and DMOD is a "Division by Zero" error when the value of the second argument is zero.

12 This function combines the magnitude of its first argument with the sign of the second. If the value of the first argument is zero, the result is zero, which is neither positive nor negative.

13 The value of the argument of the LEN function need not be defined at the time the function reference is executed.

14 INDEX(a1,a2) returns an integer value indicating the starting

position within the character string a1 of a substring identical to string a2. If a2 occurs more than once in a1, the starting position of the first occurence is returned.

If a2 does not occur in a1, the value zero is returned. Note that zero is returned if LEN(a1) < LEN(a2).

15 The REAL function for real-part extraction is the same specific function that is selected when the generic function REAL is given a complex*8 argument.

The DREAL function for real-part extraction is the same specific function that is selected when the generic function DBLE is given a complex*16 argument.

REAL and DREAL for real-part extraction could not be passed as arguments in FORTRAN 77 because they are specific type-conversion functions. To provide symmetry with AIMAG and DIMAG imaginary-part extraction, which can be passed, F77 allows REAL and DREAL to be passed as arguments.

16 A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.

17 The value of the argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT and CDSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

18 The value of the argument of ALOG, DLOG, ALOG1Ø, and DLOG1Ø must be greater than zero. The value of the argument of CLOG and DLOG must not be (Ø.,Ø.). The result of CLOG and DLOG is the principal value, i.e. the range of the imaginary part of the result is -pi < imaginary part <= pi. The imaginary part of the result is pi only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

19 All angles are expressed in radians.

2Ø The result will be expressed in radians.

21 The absolute value of the argument of SIN, DSIN, COS, DCOS, TAN, and DTAN is not restricted to be less than 2*pi.

22 The absolute value of the argument of ASIN and DASIN must be less than or equal to one. The range of the result is: -pi/2 <= result <= pi/2.

23 The absolute value of the argument of ACOS and DACOS must be less than or equal to one. The range of the result is: Ø <= result <= pi.

24 The range of the result for ATAN and DATAN is: -pi/2 <= result <= pi/2. If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and pi if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is pi/2. The arguments must not both have the value zero. The range of the result for ATAN2 and DATAN2 is: -pi < result <= pi.

25 LGE(a1,a2) returns the value true if a1=a2 or if a1 follows a2 in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LGT(a1,a2) returns the value true if a1 follows a2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLE(a1,a2) returns the value true if a1 = a2 or if a1 precedes a2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLT(a1,a2) returns the value true if a1 precedes a2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

The result-type for LGE, LGT, LLE, and LLT will be LOGICAL*2 in a program unit compiled with -LOGL, and LOGICAL*4 in a program unit compiled with -LOGS.

26 AND, OR, and XOR perform the bitwise logical function named on a list of long and short integers. The result will be a long integer if any argument is long; otherwise it will be a short integer.

When short and long integers are mixed, the short integers will be sign-extended, not zero-extended.

27 Performs a bitwise logical NOT function (ones complement) on a long or short integer. The result has the type of the argument.

28 LS and RS take two arguments; each argument may be either a long or a short integer. These arguments are called ARG1 and ARG2 in the following.

LS shifts ARG1 to the left by the number of bits specified in ARG2. The result has the type of ARG1 - that is, no type-change occurs. Vacated places are filled with zeroes. If ARG2 is not

positive, no shift occurs.

RS is identical to LS, except that the shift is to the right.

29 SHFT is similar to LS and RS, except that it can shift in either direction, and can perform two shifts rather than one. The additional shift occurs if a third integer argument, ARG3, is given.

If ARG2 is negative, the shift is to the left; if it is positive, the shift is to the right; if it is zero, no shift occurs.

If ARG3 appears, the shift specified by it will be carried out after the shift specified by ARG2 is complete. The rules are the same as for the ARG2 shift.

30 LT takes two arguments; each argument may be either a long or a short integer. These arguments are called ARG1 and ARG2 in the following.

LT preserves the left ARG2 bits of ARG1, and sets the rest to zero (left truncation). The result has the type of ARG1 - that is, no type-change occurs. If ARG2 is <= 0, no bits are preserved.

RT is identical to LT, except that the right ARG2 bits are preserved.

31 LOC operates on an item of any data type except CHARACTER and LOGICAL*1. The result is an INTEGER*4 value representing the memory address where the first byte of the data item is located.

32 An integer result produced by this function will be INTEGER*2 in a program unit compiled with -INTS, and INTEGER*4 in a program unit compiled with -INTL.

33 When this function operates on integers, the arguments may be a mixture of INTEGER*2 and INTEGER*4. The result will have the type of the longest argument.

A special case arises when IABS, MOD for integers, ISIGN, or IDIM is passed as an actual argument to a subprogram. In this case, the invoking program unit has no opportunity to examine the argument list on which the function will operate. Therefore it cannot select the version of the function that will implement the above rule. For compatibility with the FORTRAN 77 standard, the following rule is used instead:

When IABS, MOD for integers, ISIGN, or IDIM is passed as an actual argument to a subprogram, the function passed will accept and produce INTEGER*4 values if the invoking program unit was compiled with -INTL, and INTEGER*2 values if it was compiled with -INTS. This is the only case in which integer types cannot be mixed in the argument list of an integer intrinsic function.

34 This function cannot be passed as an argument to a subprogram.

35 The specific function accepting the COMPLEX*16 data type is an F77 extension.

36 This function is an F77 extension.

SECTION 7

USING THE F77 COMPILER

INTRODUCTION

Prime's FORTRAN 77 compiler accepts a source program meeting the FORTRAN 77 or F77 standard. It can output a source listing, an error listing, an object file, and various messages. Errors are printed at the terminal as the compiler detects them.

This section tells:

- How to invoke the compiler
- How to specify options to the compiler
- The significances of the various messages that are printed during compilation
- The meanings of the various compiler options

INVOKING THE COMPILER

The FORTRAN 77 Compiler is invoked by the F77 command to PRIMOS:

F77 pathname [-option 1] [-option 2] . . . [-option n]

pathname The pathname of the FORTRAN 77 source program to be compiled.

options Mnemonics for the options controlling compiler functions.

All mnemonic options must be preceded by a dash "-". Example:

F77 TEST1 -RANGE -DEBUG -LISTING

will cause TEST1 to be compiled with the options given.

COMPILER ERROR MESSAGES

For each error encountered in the program, an error message will be printed at the terminal and in the source listing if one exists. The general format of an error message is:

ERROR xxx SEVERITY y BEGINNING ON LINE zzz
explanation

xxx	Error Code
y	Level of severity
zzz	Line number where error begins
explanation	Description of the error, and possible remedies.

The significance of the severity code is:

Severity	Description
1	Warning.
2	Error that has been corrected.
3	Uncorrected error - prevents optimization and code generation.
4	Error that prevents further compilation.

FORTRAN 77 Error Messages are self-explanatory. They are not listed in this guide, since such a listing could only repeat information already given in the individual messages.

END-OF-COMPILATION MESSAGE

After the compilation process is complete, the compiler prints an end-of-compilation message at the terminal. Its format is:

xxxx ERRORS (F77-REV ZZ.Z)

xxxx	The number of compilation errors (0000 indicates a successful compilation)
ZZ.Z	The current revision number of the F77 compiler

After compilation, control returns to the PRIMOS level.

COMPILER OPTIONS

The available compiler options can be categorized as follows:

- Specify the source file
- Specify the existence and contents of the source listing
- Specify the error and statistics files
- Specify the existence and properties of the object code

Compiler options generally come in pairs: for each one, there is a converse option having the opposite effect. Most option pairs direct the compiler to do/not-do some action. A few present a choice between two actions. One member of each pair is always the default.

In the following list, each option is given along its converse. The Prime-supplied default is underlined. Commonly used options are marked with an asterisk: new users should skip over the unasterisked options.

Some options require an argument in addition to the option specification. The argument follows the option, and is not preceded by a dash. Options may be given in any order.

Table 7-1 lists the options in the order that they are discussed below. At the end of this section, Table 7-2 lists them alphabetically with their abbreviations, to provide a quick reference.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with the default options -INTL, -LOGL, -NODOl, and -DYNM.

Specify the Source File

The source file is usually designated by pathname immediately after the F77 command. Alternatively, it may be given in an option. Lowercase letters in the source can be automatically mapped to uppercase before compilation.

Table 7-1. Compiler Options

Specify the Source File

-S and -I	Give name of source file
-UPCASE / -LCASE	Convert source file to upper case

Specify the Existence and Contents of the Source Listing

*	-L [argument]	Controls existence of listing file
*	-XREF / -NOXREF	Cross reference in source listing
	-EXPLIST / -NOEXPLIST	Assembly code in source listing
	-OFFSET / -NOOFFSET	Offset map in source listing

Specify the Error and Statistics Files

-ERRLIST / -NOERRLIST	Errors-only listing file.
-ERRTTY / -NOERRTTY	Error messages at terminal
-SILENT / -NOSILENT	Suppress Warning Messages
-DCLVAR / -NODCLVAR	Flagging of undeclared variables
-STATISTICS / -NOSTATISTICS	Print compilation statistics

Specify the Existence and Properties of the Object Code

*	-B [argument]	Controls existence of object file
	-BIG / -NOBIG	Controls dummy array handling
	-DYNM / -SAVE	Controls dynamic/static allocation
	-INTL / -INTS	Controls type INTEGER storage default
	-LOGL / -LOGS	Controls type LOGICAL storage default
	-64V / -32I	Controls addressing mode
*	-DEBUG / -NODEBUG	Controls generation of debugger code
	-DO1 / -NODO1	Controls type of DO-loops
*	-OPTIMIZE / -NOOPTIMIZE	Controls optimization
	-PRODUCTION / -NOPRODUCTION	Controls generation of debugger code
*	-RANGE / -NORANGE	Inserts range-checking code

* Indicates options most useful to new users.

Prime-supplied defaults are underlined.

▶ -S[OURCE] pathname and -I[NPUT] pathname

Either of these can be used to designate the source file to be compiled, as an alternative to naming the file immediately after the F77 command. The following are equivalent:

```
F77 pathname -DYNM -INTL

F77 -DYNM -INTL -I pathname

F77 -INTL -S pathname -DYNM
```

The pathname must not be designated more than once.

▶ -UPCASE / -LCASE

Controls mapping of lowercase to uppercase letters in a source program.

UPCASE: Any lowercase letters in the source will be treated as uppercase by the compiler, except in Hollerith and CHARACTER constants.

LCASE: Lower and uppercase letters remain distinct. Keywords must be in upper case only.

Specify the Existence and Contents of the Source Listing

The F77 compiler's primary output to the programmer is the source listing. When the -L option is given, a basic source listing is created, containing:

- Date and time of compilation
- Options in effect
- Source text
- External entry points
- Symbol-Table Listing
- List of errors

Additional options can be given, to cause additional data to be inserted into the source listing: a cross reference, offset map, or pseudo-assembly code listing may be included. If such an option is given but no source listing was specified, -L YES will be assumed.

▶ * -L[ISTING] [argument]

Controls creation of the source listing file. The argument may be:

pathname	Listing will be written to the file pathname.
YES	Listing will be written to a file named L_program, where program is the name of the source file.
TTY	The listing will be printed at the user terminal.
SPOOL	The listing will be spooled directly to the line printer. Default SPOOL arguments are in effect.
NO	No listing file will be generated.

When no -L option is given, -L NO will be presumed. When -L is given with no argument, -L YES will be presumed.

▶ * -XREF / -NOXREF (Implies -L)

Controls generation of a cross reference

XREF: A cross reference will be appended to the source listing. A cross reference lists, for every variable, the number of every line on which the variable was referenced.

NOXREF: No cross reference will be generated.

▶ -EXPLIST / -NOEXPLIST (Implies -L)

Inserts a pseudo-assembly code listing into the source listing.

EXPLIST: Each statement in the source will be followed by the pseudo-PMA (Prime Macro Assembler) statements into which it was compiled. For information on PMA, see The Assembly Language Programmer's Guide, FDR3Ø59.

NOEXPLIST: No assembler statements are printed.

▶ -OFFSET / -NOOFFSET (Implies -L)

Appends an offset map to the source listing.

OFFSET: An offset map is appended to the source listing. For each statement in the source program, the offset map gives the offset in the object file of the first machine instruction generated for that statement.

NOOFFSET: No offset map is created.

Specify the Error and Statistics Files

All error messages are automatically included in the source listing if one is created. A separate errors-only file can be created. All error messages are normally printed at the terminal; this can be prevented. All warning messages can be suppressed. Undeclared variables are usually compiled without comment from the compiler, but they can be flagged as errors.

Compiler statistics can be printed at the terminal after each phase of compilation, but not to a user file other than a COMOUTPUT file.

▶ -ERRLIST / -NOERRLIST

Controls generation of an errors-only file.

ERRLIST: A listing file will be generated, named as described under -L YES, which contains only the error messages for the compilation. This option has no effect when a full source listing is specified or implied.

NOERRLIST: No such file is generated. Does not override -L.

▶ -ERRTTY / -NOERRTTY

Controls printing of error messages at the terminal.

ERRTTY: Error messages will be printed at the terminal during compilation.

NOERRTTY: No error messages will be printed. They will still be included in the source listing file, if any.

▶ -SILENT / -NOSILENT

Suppresses WARNING messages.

SILENT: Level 1 Error Messages will not be printed at the terminal, and will be omitted from any listing file.

NOSILENT: Level 1 Error Messages are retained.

▶ -DCLVAR / -NODCLVAR

Controls flagging of undeclared variables.

DCLVAR: A warning will be generated for any variable that is used in the program, but not included in a type-statement.

NODCLVAR: No such warning will be generated.

▶ -STATISTICS / -NOSTATISTICS

Controls printout of compiler statistics.

STATISTICS: A list of compilation statistics is printed at the terminal after each phase of compilation. For each phase the list contains:

- DISK: Number of reads and writes during the phase, excluding those needed to obtain the source file.
- SECONDS: Elapsed real time.
- SPACE Internal buffer space used for symbol table, in 16K byte units.
- PAGING Disk I/O time.
- CPU CPU time in seconds, followed by the clock time when the phase was completed.

NOSTATISTICS: Statistics are not printed.

Specify the Existence and Properties of the Object Code

For a given source program, the compiler can produce a variety of object programs or none at all, depending on the options given. The areas open to programmer control are:

- Creation of the object file
- Storage allocation and addressing
- Compiler augmentation of the object code

Creation of the Object File: The -B option controls the existence and naming of the object file, but not the properties it will have.

▶ * -B[INARY] [argument]

The argument may be:

pathname Object code will be written to the file pathname.

YES Object code will be written to the file named B_program, where program is the name of the source file.

NO — No binary file will be created. Specified when only a syntax check is desired.

When no -B option is given, or -B without an argument is given, -B YES will be presumed.

Storage Allocation and Addressing: By giving appropriate options, the programmer can cause compiled subprograms to accept array arguments longer than a segment, and can determine the data storage mode (static or dynamic) and the addressing mode (64V or 32I) to be used in the object file.

Any type INTEGER or LOGICAL data item which does not have a *(length) explicitly declared in a type-statement will be assigned a default length by the compiler. The defaults can be changed by two compiler options.

▶ -BIG / -NOBIG

Determines code generated for dummy array references in a subprogram.

BIG: A dummy array can become associated with any array.

NOBIG: A dummy array can become associated only with an array that does not cross a segment boundary.

See ARRAYS AS ARGUMENTS in Section 5 for details.

▶ -DYNM / -SAVE

Determines data-storage mode: dynamic or static.

Dynamic-storage variables are kept in the stack. At each call to a subprogram, space for its dynamic variables is allocated. At RETURN, the space is freed, and the data lost.

Static-storage variables are kept in the link frame. They exist at all times, and maintain their values until the program terminates.

All variables mentioned in a SAVE statement or initialized in a DATA or type-statement are static. All variables in COMMON are static. The -DYNM / -SAVE option affects only variables not SAVEd or in COMMON.

DYNM: All variables not SAVEd or in COMMON are allocated dynamic storage.

SAVE: All variables are allocated static storage.

DYNM is used principally to save space in user memory.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -DYNM.

▶ * -INTL / -INTS

Determines default lengths for type INTEGER data items whose length is not explicitly declared.

INTL: Every type INTEGER data item, including constants and parameters, will be compiled as INTEGER*4 unless the item has been explicitly declared INTEGER*2 in a type-statement.

INTS: Every such data item will become INTEGER*2 unless it is explicitly declared INTEGER*4. A constant will become INTEGER*4 under -INTS if:

1. Its value lies outside the INTEGER*2 range.

2. Its representation, including leading zeroes, contains more than 5 decimal or 6 octal digits.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -INTL.

▶ -LOGL / -LOGS

Determines default lengths for type LOGICAL data items whose length is not explicitly declared, and for the logical constants.

LOGL: Data items declared type LOGICAL without no *(length) specified become LOGICAL*4.

LOGS: Such data items become LOGICAL*2.

Logical variables can also be declared LOGICAL*1, but cannot be caused to default to that length.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -LOGL.

▶ -64V / -32I

These determine the addressing mode to be used in the object code. 64V is a segmented virtual addressing mode for 16-bit machines. 32I is a segmented virtual mode which takes maximum advantage of the 32-bit architecture of Prime's more advanced models (P45Ø and up). R and S modes (relative and sectored addressing) are not available for F77.

Augmented Object Code

When no augmented-code options are given, the source program is compiled statement by statement, and the resulting object code becomes the object file. Alternatively, the compiler can optimize the object code, and can add additional code to provide range checking, one-trip DO-loops, or the capacity to run under the symbolic debugger.

▶ * -DEBUG / -NODEBUG

Controls generation of code for the debugger.

DEBUG: The object file is modified so that it will run under the symbolic debugger. Execution time is increased. The code generated will not be optimized.

NODEBUG: No debugger code is generated.

▶ -DO1 / -NODO1

Controls the type of DO-loop which the compiler will produce.

DO1: All DO-loops will be of the FORTRAN type, and all FTN restrictions on DO-loops will be enforced. This option is provided for upward compatibility of FTN programs.

NODO1: F77 DO-loops will be produced. These are described in Section 3; they differ significantly from those in FTN.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -NODO1.

▶ * -OPTIMIZE / -NOOPTIMIZE

Controls the optimization phase of the compiler.

OPTIMIZE: The object code will be optimized. Optimized code runs more efficiently than non-optimized code, but takes somewhat longer to compile.

NOOPTIMIZE: Optimization does not occur.

▶ -PRODUCTION / -NOPRODUCTION

Alternative option controlling code for the debugger.

PRODUCTION: Similar to DEBUG, except that the code generated will not permit insertion of statement break points. Execution time increases less than when DEBUG is given.

NOPRODUCTION: Production-type code is not generated.

▶ * -RANGE / -NORANGE

Controls error checking for out-of-bounds values of array subscripts and character substring indexes.

RANGE: Error-checking code is inserted into the object file. Should an array subscript or character substring index take on a value outside the range specified when the referenced data item was declared, an error will be generated.

NORANGE: Out-of-bounds values will not generate an error message. The program will be more vulnerable to errors, but will execute more quickly.

OPTION ABBREVIATIONS

The F77 compiler options may be abbreviated, as follows:

The abbreviations -L, -B, -I, and -S stand for -LIST, -BINARY, -INPUT, and -SOURCE, respectively, regardless of what other abbreviations are used.

Except where the above rule takes precedence, the abbreviation for any compiler option is the shortest string of leftmost characters from the option's name that uniquely identify the option. Any number of additional characters, up to the complete name, may also be given.

These rules produce the abbreviations shown in Table 7-2. The table is also intended to provide a quick alphabetical reference for those already familiar with the compiler options.

Table 7-2. Summary of Compiler Options and Abbreviations.
(Defaults are underlined.)

Option	Abbreviation	Significance
-BIG	-BIG	Boundary-spanning code
-BINARY	-B	Creation of object file
-DCLVAR	-DC	Flag undeclared variables
-DEBUG	-DE	Debugger code
-DO1	-DO	FTN DO-loops
-DYNM	-DY	Dynamic storage default
-ERRLIST	-ERRL	Create errors-only file
-ERRTTY	-ERRT	Write errors to terminal
-EXPLIST	-EX	Expanded source listing
-INPUT	-I	Designate source file
-INTL	-INTL	Long integer default
-INTS	-INTS	Short integer default
-LCASE	-LC	No source-file case conversion
-LIST	-L	Creation of source listing
-LOGL	-LOGL	Long logical-data default
-LOGS	-LOGS	Short logical-data default
-NOBIG	-NOB	No boundary-spanning code
-NODCLVAR	-NODC	Don't flag Undeclared Variables
-NODEBUG	-NODE	No debugger code
-NODO1	-NODO	F77 DO-Loops
-NOERRLIST	-NOERRL	No errors-only file
-NOERRTTY	-NOERRT	No errors to terminal
-NOEXPLIST	-NOEX	No expanded source listing

Table 7-2. Summary of Compiler Options and Abbreviations (continued). (Defaults are underlined.)

Option	Abbreviation	Significance
-NOOFFSET	-NOOF	No offsets in source listing
-NOOPTIMIZE	-NOOP	Don't optimize object code
-NOPRODUCTION	-NOP	No production code
-NORANGE	-NOR	No range checking
-NOSILENT	-NOSI	Don't suppress warning messages
-NOSTATISTICS	-NOST	Don't print statistics
-NOXREF	-NOX	Don't generate cross reference
-OFFSET	-OF	Offsets in source listing
-OPTIMIZE	-OP	Optimize object code
-PRODUCTION	-P	Generate production code
-RANGE	-R	Check subscript ranges
-SAVE	-SA	Static storage default
-SILENT	-SI	Suppress warning messages
-SOURCE	-S	Designate source file
-STATISTICS	-ST	Print compiler statistics
-UPCASE	-U	Convert to uppercase
-XREF	-X	Generate cross-reference
-32I	-3	Produce 32I mode code
-64V	-6	Produce 64V mode code

SECTION 8

OPTIMIZING F77 PROGRAMS

OPTIMIZING F77 PROGRAMS

This section presents some programming hints for improving the performance of F77 programs. Some of them are merely reminders of good coding practice; others take advantage of implementation techniques in the F77 compiler. All offer some speedup in program execution.

Referencing Multi-Dimensional Arrays

Reference memory as sequentially as possible. For multi-dimensional arrays, the leftmost subscript varies the fastest in FORTRAN 77. When addressing large portions of an array, paging time and working set size can be significantly reduced by indexing the leftmost subscript the fastest (e.g., in the innermost loop). Thus,

```
      DIMENSION ARRAY (100,100)
      DO 20 I = 1, 100
         DO 10 J = 1, 100
            ARRAY (J, I) = 3.0
10          CONTINUE
20    CONTINUE
```

is more efficient than accessing the array as ARRAY (I, J) = 3.0.

If the program can be coded CLEANLY without multi-dimensional arrays, memory addressing can be more efficient. For each dimension over one, this saves one 'multiply' per effective address calculation; i.e., number-of-multiplies = number-of-dimensions - 1. For instance, the example above could be written as:

```
      DIMENSION ARRAY (100,100)
      DIMENSION INITARRAY (1)
      EQUIVALENCE (ARRAY(1,1), INITARRAY(1))

      DO 10 I = 1, 10000
         INITARRAY(I) = 3.0
10 CONTINUE
```

saving considerable CPU time.

Load Sequence and Memory Allocation

Paging time can be significantly reduced by loading subprograms by frequency of use (rather than, say, alphabetically). The main program must always be loaded first for SEG to work properly.

A suitable loading scheme would allocate memory as:

```
MAIN
 ..
 ..
END


 .
 .          most common subroutines
 .


 .
 .          occasionally used subroutines
 .


 .
 .          infrequently used subroutines
 .
```

Paged memory fragmentation can be reduced by loading routines on page boundaries using SEG's P/LO command.

In subroutine libraries, the top down tree structure must be preserved if 'reset force load' is in use.

This ordering method may also be used to order COMMON blocks in memory by frequency of use.

See The LOAD and SEG Reference Guide for details on these recommendations.

Function Calls

Eliminate redundant invocations of user-supplied functions. For example:

```
TEMP = FUNC(X)
A = TEMP * TEMP
```

is faster than:

```
A = FUNC(X) * FUNC(X)
```

Make sure that the function has no side effects which might modify the argument(s) or anything else in the environment.

This practice is not necessary with intrinsic functions unless optimization of the program unit is prevented by the -NOOPTIMIZE

compiler option, because the F77 optimizer eliminates redundant intrinsic function calls.

Input/Output

Significant speed improvement in raw data transfers can be achieved by using the equivalent IOCS or file system routine instead of formatted input/output. For example:

```
      INTEGER TEXT(40)
      READ (5, 20, END= 99) TEXT
   20 FORMAT(40A2)
```

is slower than

```
      INTEGER TEXT(40)
      CALL RDASC(5, TEXT, 40, $99)
```

but the fastest yet is...

```
      INTEGER TEXT(40), CODE
      CALL RDLIN$(1, TEXT, 40, CODE)

      IF(CODE .NE. 0)      /* Any error?
     * GOTO 99             /* Yes, go process error.
```

There are also routines for reading/writing octal, decimal, and one-unit hexadecimal numbers from/to the terminal. For example, CALL TIHEX(N) will read a hexadecimal integer from the terminal into the short integer named N. For printing out text efficiently, use the TNOU/TNOUA routines. See The PRIMOS Subroutines Reference Guide for more specific information about these lower level routines.

Statement Sequence

The compiler can do register tracking, but cannot reorder statements. For example, given the sequence:

```
A = B
X = Y
R = B
```

the generated code is:

```
LDA B
STA A
LDA Y          (6 instructions long)
STA X
LDA B
STA R
```

If the source is rearranged to:

```
A = B
R = B
X = Y
```

the generated code is reduced to:

```
LDA B
STA A
STA R          (5 instructions long)
LDA Y
STA X
```

Parameter Statements

Initializing named constants via PARAMETER statements allows the compiler to perform constant-folding optimizations, resulting in faster execution of statements using the named constants. The compiler does not fold normal variables initialized by DATA statements into constants.

Inefficient Library Calls

Some applications library routines are not optimized for time-critical operations. The get and store character routines (GCHR$A, etc.) are convenient, but comparatively slow. Some applications library routines are by definition slow, because they use lower-level routines which can more efficiently be called directly. Avoid using the MAX and MIN functions when execution time must be minimized.

Applications library subroutines are designed to perform acceptably at any task for which they might be called. When one particular task is often required in a program, a user-supplied routine which is maximally efficient at that one task can be substituted. See Section 5 and the EXTERNAL Statement in Section 3.

Remember the 8Ø/2Ø rule, which states: "8Ø percent of a program's time is spent in 2Ø percent of the code" (exact numbers subject to debate). Therefore, standard library routines are adequate in the non-time-critical 8Ø percent of the program.

Statement Functions and Subroutines

Use statement functions instead of FUNCTION subprograms when practical. This eliminates a lengthy PCL/PRTN sequence. Try to minimize the number of arguments passed to and from a statement function, function, or subroutine.

Integer Divides

When dividing a non-negative integer by a power of two, use the RS (right shift) binary intrinsic function. For example:

```
I = RS(J, 3)
```

Is much faster than:

```
I = J / 8
```

Use of the Compiler's -DYNM option

F77 programs run faster, better, and cleaner when local variables are placed in the stack through the -DYNM option (the default). These variables are not guaranteed to be valid after a return.

Conclusion

These are some of the more common guidelines to keep in mind while programming in Prime FORTRAN 77. If you keep these ideas in mind while writing, or while 'fine tuning' FORTRAN 77 programs, your programs will generally be smaller and faster. Some of these rules are not necessarily permanent. As Prime FORTRAN 77 evolves more and more optimizations, the user will have more freedom to choose coding styles.

Generally it is easier to apply these techniques at initial coding time, as opposed to 'going back and optimizing'. While some of these changes can be done easily with a few Editor tricks, others may require extensive changes to the source code.

Only specific techniques which can be described fairly briefly are mentioned in this section. Many other examples of good programming practice, and an excellent discussion of the more general aspects of good programming, appear in the following text:

Kernighan and Plauger, The Elements of Programming Style, McGraw-Hill, 1974

APPENDIX A

CONVERTING FTN PROGRAMS TO F77

The conversion of FTN programs to F77 is in general straightforward. The techniques required for such a conversion are described in this section.

The relative simplicity of converting FTN programs to F77 results from two factors:

- The designers of FTN made use of preliminary documents released by ANSI during the development of FORTRAN 77. The information in these documents was used to make FTN's extensions to FORTRAN 66 identical to those of the future FORTRAN 77 wherever this could be accomplished without violating the FORTRAN 66 standard.

- F77 includes all FTN constructs, except the obsolete TRACE statement, that are absent from but compatible with FORTRAN 77.

The result is that many FTN program units can be compiled in F77 with no changes. Most of the rest can be converted with only minor changes.

The rare program unit which cannot easily be converted to F77 can usually be left in FTN form and called by other units that are written in F77. See USING AN FTN PROGRAM UNIT IN AN F77 PROGRAM, below.

METHODOLOGY OF PROGRAM CONVERSION

Any project converting FTN programs to F77 should have available:

- This guide

- The Prime User's Guide

- The FORTRAN IV Reference Guide, FDR3Ø57

- The ANSI Standard for FORTRAN 77

Conversion of a program to F77 need not be an all-or-none process. Due to the similarity of FTN and F77, each unit of an FTN program can be dealt with separately when the program as a whole is converted.

The first step in converting an FTN program unit to F77 is to compile it in F77 and see what, if any, error messages result. Due to the detailed and often prescriptive information given by an F77 error message, the messages produced should give a fairly complete picture of the changes needed.

The second step is to search the FTN program unit for constructs which

are common to and syntactically the same in FTN and F77, and hence generate no syntax errors, but which have different requirements or results in the two languages due to differences between the ANSI standards. Such constructs are called "optionally acceptable FTN constructs" and "reimplemented FTN constructs." These terms are defined, and all such constructs are described, below under PRODUCING AN F77-COMPATIBLE PROGRAM UNIT.

The first and/or second steps should be iterated until the program unit compiles correctly with all optionally acceptable and reimplemented constructs dealt with as necessary.

The third step is a thorough check of the converted program unit. Before it is accepted as correct, it should pass the same tests it was required to pass before being accepted in its original version.

Caution

> The fact that a program unit compiles without error in F77 does not mean it will produce the same results in F77 that it did in FTN. Identity of results can be achieved only if all optionally acceptable and reimplemented constructs have been correctly dealt with.

DEGREES OF PROGRAM UNIT CONVERSION

Conversion of a program unit to F77 is not an all-or-none matter. Three degrees of conversion of an FTN program unit can be distinguished:

- The unit may be left in FTN, but reference and be referenced by other units that are in F77. This conversion is contextual - the unit per se remains an FTN program unit.

- The unit may be recompiled in F77, but retain certain optionally acceptable FTN constructs that violate the FORTRAN 77 standard. The F77 compiler will compile them correctly only if it is invoked with appropriate options, as described below. A program unit of this type is termed an F77-compatible program unit.

- The unit may be completely converted to standard-conforming F77. It is then termed an F77-standard program unit.

There is no need for all units of a converted program to be converted to the same degree.

USING AN FTN PROGRAM UNIT IN AN F77 PROGRAM

An FTN program unit may reference and be referenced by an F77 program unit. The comments in Section 1 under INTERFACE WITH OTHER LANGUAGES apply. A few additional restrictions must also be kept in mind:

- An F77 function returning a COMPLEX*8 value cannot be referenced by an FTN program unit, nor can an FTN function returning a COMPLEX*8 value be referenced by an F77 program unit.

- Data of types which exist in F77 but not in FTN cannot be passed as arguments.

- An F77 subroutine cannot use the F77 alternate return mechanism (i.e. RETURN (expression)) if it will be called by an FTN program unit. The F77 subroutine must use the old-style alternate mechanism (i.e. GO TO (dummy variable)).

Any program unit for which no modifications to use the added power of F77 are contemplated, and which can be invoked by an F77 program unit, can be left in FTN indefinitely.

No F77 program unit can reference or be referenced by any program unit that was compiled in R-mode. An FTN unit in R-mode must be recompiled into V-mode before it can become part of an F77 program. A few rarely used R-mode FTN constructs are not available in V-mode. See below under Unsupported FTN Constructs.

PRODUCING AN F77-COMPATIBLE PROGRAM UNIT

The information needed to convert an FTN program unit to an F77-compatible program unit falls into four categories:

- Constructs that are compiled differently by the FTN and F77 compilers, but which will be compiled in the FTN manner by the F77 compiler if the compiler is invoked with appropriate options (Optionally acceptable FTN constructs).

- Constructs that are compiled differently by the FTN and F77 compilers, and which cannot be compiled in the FTN manner by the F77 compiler (Reimplemented FTN constructs).

- Constructs that exist in FTN but not in F77 (Unsupported FTN constructs).

- Constructs that exist in FTN and are not part of FORTRAN 77, but have been added to F77 for compatibility (Obsolete FTN constructs).

Optionally Acceptable FTN Constructs

The various compiler options mentioned below are fully defined in Section 7.

The optionally acceptable FTN constructs, and their F77 versions, are as follows. In each case, the F77 version conforms to the FORTRAN 77 standard, while the FTN version does not.

FTN DO-Loops: An FTN DO-loop always executes once, and permits extended DO-ranges, while an F77 DO-loop can execute zero times and forbids extended DO-ranges. This difference can be insidious because all FTN DO-loops are syntactically correct in F77. There are also other differences, but these do not affect program unit conversion. The two types of loop are fully compared under the DO Statement in Section 3.

To cause the F77 compiler to produce FTN-type DO-loops, invoke it with the -DO1 option.

Short Integers: In FTN, the type INTEGER without a *(length) specification is synonymous with INTEGER*2 (short integer) and integer constants are stored as INTEGER*2 unless they are too big or contain too many digits. (See Section 2.) In F77, INTEGER is synonymous with INTEGER*4 (long integer) and integer constants are stored as INTEGER*4.

To cause the F77 compiler to produce short integers in the manner of the FTN compiler, invoke it with the -INTS option.

The FTN compiler has the -INTL option, which causes it to treat integer data in the manner described for F77. A program unit that was normally compiled with -INTL in FTN requires no special action regarding integer data when converted to F77.

Short Logical Data: In FTN, logical data always occupies two bytes (LOGICAL*2); there is no LOGICAL*4 type. In F77, the type LOGICAL without a *(length) specification is synonymous with LOGICAL*4, and logical constants are stored as LOGICAL*4.

To cause the F77 compiler to produce short logical data (except where LOGICAL*4 has been explicitly specified) invoke it with the -LOGS option.

Static Storage Default: Both the FTN and F77 compilers offer the -DYNM/-SAVE option. In FTN, the default is -SAVE, so that all data is static. In F77, the default is -DYNM, so that all data is dynamic unless explicitly declared static. This dynamic storage property is required by the FORTRAN 77 standard.

If the correct operation of an FTN program unit is dependent on some or all of its data being static by default, the -SAVE option must be given explicitly when it is compiled in F77.

A program unit that was normally compiled with -DYNM in FTN requires no special action regarding storage class when converted to F77.

Reimplemented FTN Constructs

Most of the effort required in converting an FTN program unit to F77 will concern reimplemented constructs. Each instance of such a construct must be examined, and modified if necessary, to be sure it will produce the results desired when run under F77.

The reimplemented FTN constructs, and their F77 versions are as follows. In each case where standard-conformance is involved the F77 version conforms to the FORTRAN 77 standard, while the FTN version conforms to the FORTRAN 66 standard.

Note

Most reimplemented constructs are syntactically identical in FTN and F77. No error messages will result when such constructs are encountered: they must be found by inspecting the source code.

Listing Control: In FTN, the interaction between the compiler options that create the source listing and the in-program statements that turn source listing generation on and off is somewhat different than in F77. The two charts below illustrate the difference. Note that in F77, FULL LIST is an obsolete synonym for LIST.

FTN	-LIST NO	-LIST YES	-EXPLIST
NO LIST	NO LISTING	NO LISTING	FULL LISTING
LIST	NO LISTING	NORMAL LISTING	FULL LISTING
FULL LIST	NO LISTING	FULL LISTING	FULL LISTING

F77	-LIST NO	-LIST YES	-EXPLIST
NO LIST	NO LISTING	NO LISTING	NO LISTING
LIST	NO LISTING	NORMAL LISTING	FULL LISTING
FULL LIST	NO LISTING	NORMAL LISTING	FULL LISTING

Global Mode: FTN assigns the global mode to those names that are not explicitly typed and whose first appearance in the program follows the global mode statement. F77 assigns the global mode to all names that are not explicitly typed, whether or not they follow the global mode statement.

Intrinsic Functions: FTN treats IFIX, FLOAT, and IDINT as generic functions, not restricting their argument to a particular type. F77 provides the INT and REAL generic functions, but treats IFIX, FLOAT, and IDINT as specific functions requiring a particular type.

FTN allows LOGICAL*2 arguments in the following intrinsics: LS, RS, SHFT, LT, RT, AND, OR, NOT, and XOR. F77 allows only INTEGER*2 and INTEGER*4 arguments.

FORTRAN 77 introduces a number of new intrinsic functions. Their names may conflict with those of user-supplied subprograms. To cause such a duplicate name to refer to the user-supplied subprogram, specify it in an EXTERNAL statement. The similarly-named intrinsic will then be unavailable to that program unit.

Intrinsics in Constant Expressions: FTN allows a subset of the intrinsic functions in constant expressions. F77 does not allow this practice.

Input/Output: In FTN, an unformatted sequential file must consist of fixed-length records. In F77, such a file may consist of either fixed- or varying-length records.

In FTN, BACKSPACE works only on tape files. In F77, it will work on all formatted sequential files and on fixed-length unformatted sequential files.

In FTN, a READ or WRITE can access more than one record. In F77, a READ or WRITE always accesses a single record (slash editing excepted).

The method for increasing maximum record length has been greatly simplified in F77. Use of ATTDEV is no longer required. The F77 method is described under INCREASING MAXIMUM RECORD LENGTH in Section 4.

Extra Parentheses in I/O Statements: FTN ignores extra parentheses in I/O lists, while F77 considers them syntax errors. Prohibiting the extra parentheses prevents certain ambiguities which could otherwise arise in an I/O list.

Blanks in Format Lists: FTN allows blanks as well as commas to separate format-list descriptors. F77 ignores blanks in format lists unless they are in a character or Hollerith constant.

Slash Edit-Control Descriptor: In FTN, execution of the statement:

```
      WRITE (N,100)
  100 FORMAT (/)
```

will cause one blank record to be written. In F77, two blank records will be written.

STOP and PAUSE Statements: In FTN, the number (if any) printed by a STOP or PAUSE statement will be in octal form. F77 prints such a number in decimal.

The FTN STOP statement has no effect on I/O units. The F77 STOP statement closes any I/O units used by the program.

Unsupported FTN Constructs

The only frequently-used FTN construct not supported in F77 is the TRACE statement, which was used in conjunction with the -TRACE compiler option (also unsupported) as a debugging tool.

When assistance in debugging an F77 program is required, use the far more powerful Source Level Debugger, available from Prime as a separately priced item. For complete information on using the debugger, see The Source Level Debugger Reference Guide.

Certain specialized and rarely-used FTN constructs are dependent on FTN compiler options which are not supported by the F77 compiler. When one of these constructs has been used in an FTN program unit being converted to F77, it must be replaced with an equivalent F77 construct, or eliminated entirely. The options are:

The -32R and -64R options: A few FTN constructs are available only in R-mode: the commonly used ones are multi-level alternate returns, and variable-length argument lists. Methods that provide the same results and work in V and I mode can always be found.

The -SPO Option: The FTN constructs dependent on the -SPO option are not enumerated here, as they are of interest only to certain specialized users who need no additional information. If there is no alternative to using an -SPO construct, be sure that the program unit is otherwise callable from F77, and keep it in FTN form.

The -PBECB Option: Comments similar to those for the -SPO option apply.

Obsolete FTN Constructs

The following features of FTN are not standard in FORTRAN 77. F77 has been extended to accept them, but they are considered obsolete techniques. Do not use them in new programs.

The obsolete techniques will always produce the same results in F77 as

in FTN. They are mentioned here so that those converting FTN programs to F77 will know that, despite their nonstandard status, they can be ignored during the conversion process. They are not explained here, because they are properly part of FTN, not F77. For information on them, see The FORTRAN IV Reference Guide, FDR3057.

The obsolete features are:

- The format nOddd... for octal constants
- The ENCODE and DECODE statements for in-storage type conversion
- Hollerith strings
- Indexing a multi-dimensional array with a one-subscript reference in an EQUIVALENCE statement
- Alternate returns using a GO TO to a statement-label dummy variable
- Use of "$" instead of "*" to denote a statement label constant
- Extended DO-ranges, except when the F77 compiler is invoked with the -DO1 option for generation of FTN-type DO-loops. If an extended DO-range is present in a program compiled with -NODO1 (the default) no error will be detected, but unpredictable results will occur. See Section 7 for more on the -DO1/-NODO1 option.

PRODUCING AN F77-STANDARD PROGRAM UNIT

An F77-standard unit is a converted FTN unit which contains no optionally acceptable constructs. Such a unit must compile without errors and give the expected results when compiled with the default options -NODO1, -INTL, -LOGL, and -DYNM.

With respect to reimplemented, unsupported, and obsolete FTN constructs, the task of producing an F77-standard program unit is identical to that of producing an F77-compatible program unit.

Elimination of Optionally Acceptable Constructs

Elimination of an FTN program units's dependence on the synonymy of INTEGER with INTEGER*2 and LOGICAL with LOGICAL*2 is easy. Where INTEGER*2 or LOGICAL*2 data is specifically desired, modify or create the appropriate type-statement. Where INTEGER*4 and LOGICAL*4 will do, be sure that use of the longer data types will not cause mismatch of arguments in subprogram invocations, or unexpected results in mixed-type expressions and assignments.

Elimination of dependence on FTN-type handling of DO-loops is accomplished as follows:

1. Eliminate any extended DO-ranges. The simplest way is to substitute an appropriate subprogram invocation.

2. Where the program unit's logic is unalterably dependent on the one-trip property of the FTN DO-loop (which is only rarely the case) insert appropriate conditional statements into the source code to insure that the trip will occur.

Existing conditional statements serving only to prevent the compulsory one-trip if the DO-test is already satisfied when control reaches the loop can be left in or deleted as desired: they merely duplicate the normal action of an F77 DO-loop.

Elimination of a program unit's dependence on the -SAVE option is accomplished by naming all data items which must be static in a SAVE statement in the program unit. See the SAVE Statement in Section 3.

APPENDIX B

F77 PROGRAMMING EXAMPLE

Source File: EX.SRCE
Compiled on: 800116 at: 17:48 by: FORTRAN-77 Rev 17.2
Options: OPTIMIZE NOBIG INTL LOGL DYNM UPCASE

```
      PROGRAM DEMO                          /* PROGRAM STATEMENT */
C
C
*****************************************************************
*                                                               *
*  SAMPLE PROGRAM TO DEMONSTRATE THE VARIOUS FEATURES OF        *
*  FORTRAN 77, AND A TYPICAL F77 COMPILER SOURCE LISTING.       *
*  THIS PROGRAM IS NONSENSICAL, AND THE READER IS CAUTIONED     *
*  NOT TO TRY TO DECIPHER ITS LOGIC.                            *
*                                                               *
*****************************************************************
C
C
C*****  PARAMETER STATEMENTS
C
      INTEGER ONE,FOUR,TEN,FORTY     /* DCL TYPE BEFORE USE  */
      PARAMETER ONE = 1,
     *          FOUR = 4,
     *          TEN = 10,
     *          FORTY=TEN*FOUR     /* NOTE USE OF EXPRESSION */
C
C*****  THE CHARACTER DATA TYPE IS NEW TO FORTRAN 77.
C
      CHARACTER*4 FILE
      CHARACTER*12 FNAME, FORM*8
C
C*****  ARRAY DCL'S, USING LOWER BOUNDS AND 7 DIMENSIONS
C
      DIMENSION A(-5:5, 6, 0:9)
      DIMENSION B(1, 2, 3, 4, 5, 6, 7)
      CHARACTER C(0:FOUR, TEN)*5      /* CHAR ARRAYS ALLOWED */
                                     /* NOTE USE OF PARAMETERS */
C
C*****  LOGICAL VARIABLES - NOTE *1, *2 AND *4 FORMS.
C*****  THESE ARE NOT FORTRAN 77, BUT ARE SUPPORTED FOR
C*****  COMPATIBILITY WITH IBM.  NOTE DATA INITIALIZATION
C*****  IN A TYPE STATEMENT.
C
      LOGICAL EXISTS, OPND
      LOGICAL*1 LOG1
      LOGICAL*2 LOG2/.TRUE./, LOG2B
      LOGICAL*4 LOGICALFOUR          /* UP TO 32 CHAR NAMES */
```

```
C
C*****  COMPLEX*16 IS NOT FORTRAN 77, BUT IS AN EXTENSION FOR
C*****  COMPATIBILITY WITH IBM FORTRAN.
C
      COMPLEX*16 DCOMPVAR
C
C*****  USE OF DOUBLE PRECISION TYPE DECLARATION
C
      DOUBLE PRECISION D1, D2, D3, D4
C
C*****  EXTERNAL STATEMENT USED TO INSURE THAT AN EXTERNAL
C*****  FUNCTION WILL BE USED INSTEAD OF THE INTRINSIC.
C*****  IT COULD ALSO BE USED TO INSURE THAT ANY FUNCTION
C*****  USED WILL NOT BE MISINTERPRETED AS AN INTRINSIC EVEN
C*****  THOUGH SOME VENDOR MAY HAVE ADDED A FUNCTION OF THAT
C*****  NAME TO THE LIST OF INTRINSICS, ENHANCING PORTABILITY.
C
      EXTERNAL IFIX
C
C*****  BEGINNING OF EXECUTABLE CODE.  THE PURPOSE OF THIS
C*****  ROUTINE IS TO OPEN SOME FILES, AND THEN CHECK
C*****  THAT THE FILES WERE CORRECTLY OPENED.  THIS DEMON-
C*****  STRATES SOME OF THE NEW I/O FEATURES OF FORTRAN 77.
C
      FILE = 'FILE'      /* ASSIGN ASCII STRING TO CHAR VAR */
      SOME_NUMBER = 64.2
C
C*****  THIS IS THE MAIN LOOP
C
      DO 10 I=1,SQRT(SOME_NUMBER)*8   /* REAL EXPR FOR DO PARM
      FNAME = FILE//CHAR(I)           /* CHAR CONCATENATION */
C
C*****  NEW OPEN STATEMENT WITH KEYWORDS.
C
      OPEN (FILE = FNAME,
     *      UNIT = I,
     *      STATUS = 'UNKNOWN',
     *      ACCESS = 'SEQUENTIAL',
     *      ERR = 100)
C
C*****  NEW INQUIRE STATEMENT
C
      INQUIRE (UNIT = I,
     *         EXIST = EXISTS,
     *         OPENED = OPND,
     *         NAME = C(I+1,4),  /* EXPRESSION IN ARRAY REF */
     *         ERR = 101)
```

```
90    C
91    C*****  AN EXAMPLE OF A BLOCK IF-THEN-ELSE
92    C
93          IF (EXISTS .AND. OPND) THEN
94                   WRITE (1,*) FNAME, ' EXISTS AND IS OPENED'
95                                      /* LIST DIRECTED I/O WITH */
96                                            /* CHAR CONSTANT */
97               ELSE
98                   PRINT *, FNAME, ' NOT OPENED, NO ERROR RAISED'
99                                         /* NEW PRINT STATEMENT */
100              END IF
101   10     CONTINUE
102         GO TO 1000
103   C
104   C*****  END OF MAIN LOOP. ERROR ROUTINES FOLLOW.
105   C
106   100    WRITE (1, '(A, A, A, I3)')'ERROR ON OPEN OF ', FNAME,
107        * 'ON UNIT ', I
108                                  /* FORMAT EMBEDDED IN I/O STMT  */
109         STOP 'ERROR'
110   101    CONTINUE
111         FORM = '    (A, I3)'                    /* DEFINE FORMAT */
112         WRITE (1, FORM) 'ERROR ON INQUIRE ON UNIT ',I
113                                   /* CHAR VAR REPRESENTS FORMAT */
114         STOP 'ERROR'
115   C
116   1000   INT_RANDOM = IFIX(3.1)        /* USE EXTERNAL FUNCTION */
117   C
118   C*****  THIS NEXT CALL DEMONSTRATES THE ALTERNATE RETURN.
119   C
120         CALL ALTRET (I, $5001, $5002)
121         INT_RANDOM = 0
122         GO TO 6000
123   5001   CONTINUE                                 /* ALT RETURN #1 */
124         INT_RANDOM = 1
125         GO TO 6000
126   5002   CONTINUE                                 /* ALT RETURN #2 */
127         INT_RANDOM = 2
128   C
129   C*****  ANOTHER EXAMPLE OF THE BLOCK-IF, BUT WITH MULTIPLE
130   C*****  BRANCHES.  ALSO, MULTIPLE ENTRY POINTS OF THE
131   C*****  SUBROUTINE MULTIN ARE USED.
132   C
133   6000   IF (INT_RANDOM .EQ. 0) THEN
134                   CALL MULTIN (I, INT_RANDOM)
135              ELSE IF (INT_RANDOM .EQ. 1) THEN
136                   CALL MULT1 (I)
137              ELSE IF (INT_RANDOM .EQ. 2) THEN
138                   CALL MULT2 (INT_RANDOM)
139              ELSE
140                   INT_RANDOM = -1
141              END IF
```

```
  142   C
  143   C*****  NEXT IS AN EXAMPLE OF INTERNAL FILES.  FIRST, READ AN
  144   C*****  80 CHAR RECORD INTO BUFFER.  ASSUMING IT IS ALL
  145   C*****  NUMBERS, IT CAN BE 'READ' INTERNALLY INTO ANOTHER
  146   C*****  INTEGER VARIABLE.  INTERNAL FILES  HAVE THE SAME
  147   C*****  FUNCTIONALITY AS ENCODE/DECODE.
  148   C
  149         CHARACTER BUFFER*80            /* DEFINE INPUT BUFFER */
  150         DIMENSION IN_ARRAY(80)        /* DEFINE INTEGER ARRAY */
  151         READ (5, '(A80)') BUFFER
  152         READ (UNIT=BUFFER, FMT='(80I1)') IN_ARRAY
  153   C
  154   C*****  THIS IS AN EXAMPLE OF GENERIC TYPING OF INTRINSICS.
  155   C*****  IT IS NO LONGER NECESSARY TO USE DIFFERENT FUNCTION
  156   C*****  NAMES FOR THE SAME FUNCTION FOR DIFFERENT DATA TYPES.
  157   C
  158         D1 = 2.2                    /* DEFINE DOUBLE PREC VARS */
  159         D2 = 3.6
  160         D3 = 4.9
  161         D4 = D1 + D2 + D3
  162         SINGLE = 31.3134                          /* SINGLE PREC */
  163         SINGLE=SQRT(D1)/ABS(D2)+SQRT(D3)*SQRT(D4)/SQRT(D_SINGLE)
  164         END
```

EXTERNAL ENTRY POINTS

Entry Point	Program Unit	Line	Type
DEMO		1	ENTRY

Main Program DEMO on line 1

Name	Storage	Size	Loc	Attributes
10	CONSTANT			EXECUTABLE LABEL LINE 101
100	CONSTANT			EXECUTABLE LABEL LINE 106
101	CONSTANT			EXECUTABLE LABEL LINE 110
1000	CONSTANT			EXECUTABLE LABEL LINE 116
5001	CONSTANT			EXECUTABLE LABEL LINE 123
5002	CONSTANT			EXECUTABLE LABEL LINE 126
6000	CONSTANT			EXECUTABLE LABEL LINE 133
ONE		2H		INTEGER*4 NAMED CONSTANT 1
FOUR		2H		INTEGER*4 NAMED CONSTANT 4
TEN		2H		INTEGER*4 NAMED CONSTANT 10
FORTY		2H		INTEGER*4 NAMED CONSTANT 40
FILE	DYNAMIC	4C	000120	CHARACTER*4
FNAME	DYNAMIC	12C	000122	CHARACTER*12
FORM	DYNAMIC	8C	000130	CHARACTER*8
A	DYNAMIC	1320H	000134	REAL*4 DIMENSION(-5:5, 6, 0:9)
B	DYNAMIC	10080H	002604	REAL*4 DIMENSION(1,2,3,4,5,6,7)

C	DYNAMIC	250C	026344	CHARACTER*5 DIMENSION(0:4, 10)
EXISTS	DYNAMIC	2H	000054	LOGICAL*4
OPND	DYNAMIC	2H	000056	LOGICAL*4
LOG1	DYNAMIC	1C	026541	LOGICAL*1
LOG2	STATIC	1H	000027	INITIAL LOGICAL*2
LOG2B	DYNAMIC	1H	000060	LOGICAL*2
LOGICALFOUR	DYNAMIC	2H	000062	LOGICAL*4
DCOMPVAR	DYNAMIC	8H	026542	COMPLEX*16
D1	DYNAMIC	4H	000064	REAL*8
D2	DYNAMIC	4H	000070	REAL*8
D3	DYNAMIC	4H	000074	REAL*8
D4	DYNAMIC	4H	000100	REAL*8
IFIX	CONSTANT			INTEGER*4 FUNCTION
SOME_NUMBER	DYNAMIC	2H	000104	REAL*4
I	DYNAMIC	2H	000106	INTEGER*4
SQRT	INTRINSIC			
CHAR	INTRINSIC			
INT_RANDOM	DYNAMIC	2H	000110	INTEGER*4
ALTRET	CONSTANT			SUBROUTINE
MULTIN	CONSTANT			SUBROUTINE
MULT1	CONSTANT			SUBROUTINE
MULT2	CONSTANT			SUBROUTINE
BUFFER	DYNAMIC	80C	026552	CHARACTER*80
IN_ARRAY	DYNAMIC	2H	000112	INTEGER*4 DIMENSION(30)
SINGLE	DYNAMIC	2H	000114	REAL*4
ABS	INTRINSIC			
D_SINGLE	DYNAMIC	2H	000116	REAL*4

```
165   C
166   C
167   ****************************************************************
168   *                                                              *
169   *  THIS IS AN EXTERNAL FUNCTION OF THE SAME NAME AS THE        *
170   *  INTRINSIC IFIX, AND DOES THE SAME THING, SO AS TO           *
171   *  DEMONSTRATE THAT BY USING THE EXTERNAL STATEMENT ONE        *
172   *  CAN SUBSTITUTE ONE'S OWN VERSION OF A FUNCTION.             *
173   *                                                              *
174   ****************************************************************
175   C
176   C
177         INTEGER FUNCTION IFIX(RVAR)
178         IFIX = RVAR
179         RETURN
180         END
```

EXTERNAL ENTRY POINTS

Entry Point	Program Unit	Line	Type
IFIX		177	INTEGER*4 FUNCTION

Function IFIX on line 177

Name	Storage	Size	Loc	Attributes
RVAR	DUMMY ARG	2H	POS 1	REAL*4

```
181   C
182   C
183   ****************************************************************
184   *                                                              *
185   *  THIS SUBROUTINE DEMONSTRATES ALTERNATE RETURNS.             *
186   *                                                              *
187   ****************************************************************
188   C
189   C
190         SUBROUTINE ALTRET (I, *, *)
191         RETURN I                    /* IF I = 1, RETURNS TO 5001 */
192                                     /* IF I = 2, RETURNS TO 5002 */
193                                   /* OTHERWISE, RETURNS NORMALLY */
194         END
```

EXTERNAL ENTRY POINTS

Entry Point	Program Unit	Line	Type
ALTRET		19Ø	SUBROUTINE

Subroutine ALTRET on line 19Ø

Name	Storage	Size	Loc	Attributes
I	DUMMY ARG	2H	POS 1	INTEGER*4

```
195   C
196   C
197   ****************************************************************
198   *                                                              *
199   *  THIS SUBROUTINE IS AN EXAMPLE OF A SUBROUTINE WITH          *
200   *  MULTIPLE ENTRY POINTS.                                      *
201   *                                                              *
202   ****************************************************************
203   C
204   C
205         SUBROUTINE MULTIN (I, INT_RANDOM)
206   C
207         I = 0
208         INT_RANDOM = 13
209         RETURN
210   C
211   C*****  SECONDARY ENTRY POINT. NOTE THAT THE ARG LIST NEED
212   C*****  NOT MATCH THAT AT THE HEADER STATEMENT.
213   C
214         ENTRY MULT1 (I)
215         I = 15
216         RETURN
217   C
218   C*****  NEXT ENTRY POINT
219   C
220         ENTRY MULT2 (INT_RANDOM)
221         INT_RANDOM = INT_RANDOM**2
222         RETURN
223         END
```

EXTERNAL ENTRY POINTS

Entry Point	Program Unit	Line	Type
MULTIN		205	SUBROUTINE
MULT1	MULTIN	214	SUBROUTINE
MULT2	MULTIN	220	SUBROUTINE

Subroutine MULTIN on line 205

Name	Storage	Size	Loc	Attributes
I	DUMMY ARG	2H	POS 1	INTEGER*4
INT_RANDOM	DUMMY ARG	2H	-V-	INTEGER*4

APPENDIX C

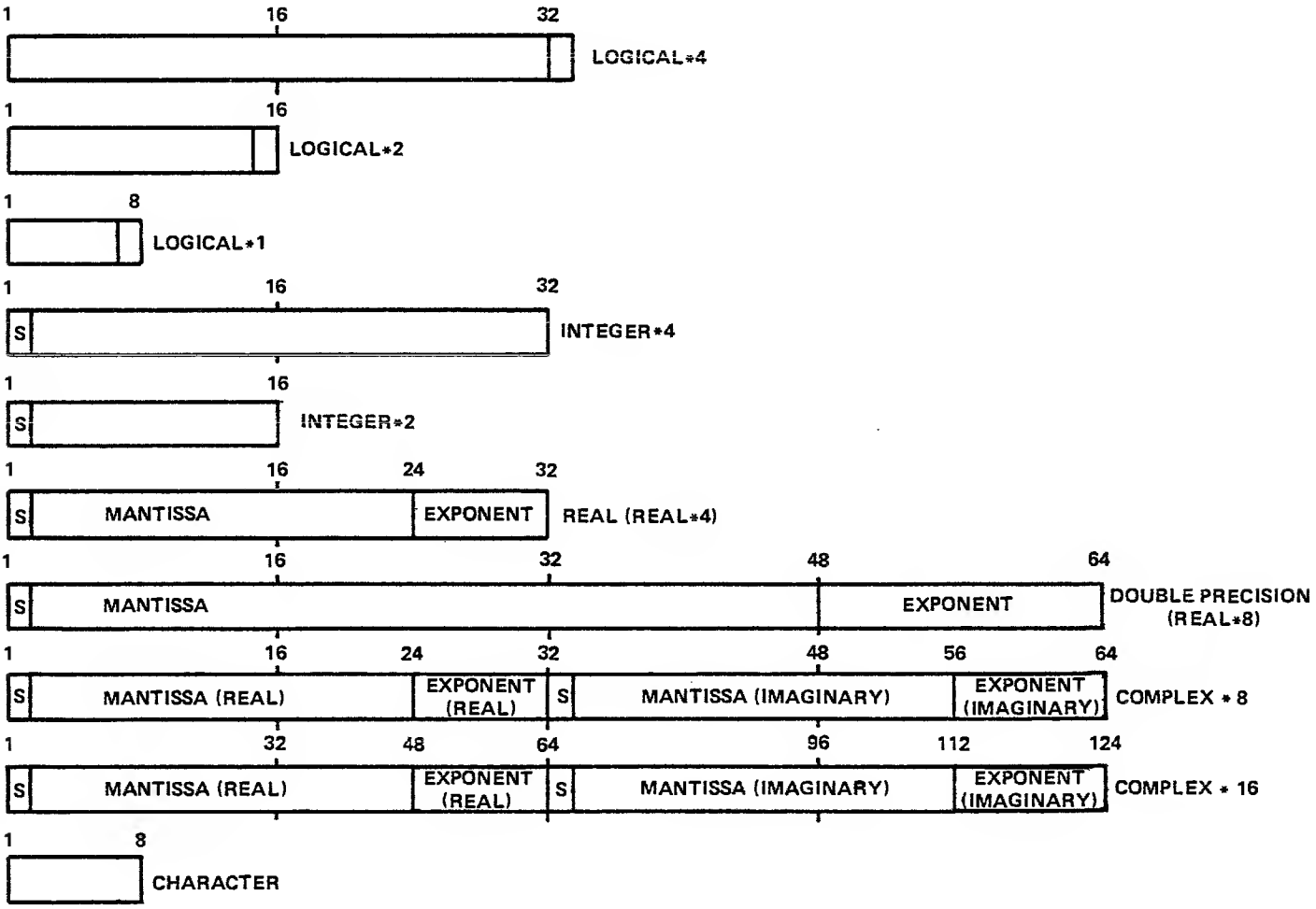
PRIME MEMORY FORMATS FOR F77 DATA TYPES

INTRODUCTION

Prime machines use a 16-bit memory word. All FORTRAN 77 data types except CHARACTER occupy either 32 bits or some multiple of 32 bits. CHARACTER data occupies one byte per character.

F77 includes the INTEGER*2, LOGICAL*2, and LOGICAL*1 types for compatibility with FTN; these occupy 16, 16, and 8 bits respectively. These types should never be used in new programs.

Figure C-1 summarizes the sizes and internal bit-usages of the F77 data types. Detailed descriptions of each type are presented below.



DATA TYPES

LOGICAL*4 32 bits. Bits 1-31=Ø Bit 32: Ø=.FALSE.
1=.TRUE.

LOGICAL*2 16 bits. Bits 1-15=Ø Bit 16: Ø=.FALSE.
1=.TRUE.

LOGICAL*1 8 bits. Bits 1-7=Ø Bit 8: Ø=.FALSE.
1=.TRUE.

INTEGER*2 16 bits. Bit 1 = sign bit. INTEGER numbers are in 2's complement representation with a value range of -32768 to 32767. These numbers in octal are '100000 and '077777 respectively. Note that -Ø=Ø, and -(-32768) = -32768.

Integer arithmetic is always exact. Integer division truncates, rather than rounds.

INTEGER*4 32 bits. Bit 1 = sign bit. Integer numbers are in 2's complement representation with a value range of -2147483648 to 2147483647. These numbers, in octal (word 1, word 2) are ('100000, '000000) and ('077777, '177777) respectively. Note that -Ø=Ø and -(-2147483648) = -2147483648.

Integer arithmetic is always exact. Integer division truncates, rather than rounds.

Caution

Explicit use of DBLE (FLOAT (I*4)) can cause the loss of the low-order 8 bits of precision. Mixed mode arithmetic, however, will not lose precision.

REAL*4 32 bits. Bit 1 = sign bit. Bits 2-24 = mantissa. Bits 25-32 = exponent. The mantissa and sign are treated as a 2's complement number and the exponent is an unsigned, excess 128, binary exponent. In general, any floating point number is represented as:

N = M * 2**(E-128)

where

$-1 \leq M < -1/2$ or $1/2 \leq M < 1$
$Ø \leq E \leq 255$

Zero is represented as M = Ø, E = Ø.

The value range, in octal (word1, word2) is:

('100000, '000377) [See Note] to ('077777, '177777)

corresponding to -1*2**(127) and (1-e)*2**(127).

The values closest to zero, in octal are:

('137777, '177400) and ('040000, '000000) [See Note]

corresponding to (-1/2+e)*2**-128 and 1/2*2**-128

Normalization ensures that bits 1 and 2 are different and is achieved by shifting left 1 bit at a time. Hence, the effective precision is between 22 and 23 bits.

Note

These numbers will cause exponent overflow if negated due to the asymmetry of 2's complement notation.

DOUBLE PRECISION 64 bits. Bit 1 = sign bit. Bits 2-48 = mantissa. Bits 49-64 = exponent. The mantissa and sign are treated as a 2's complement number and the exponent is a signed, excess 128, binary exponent. In general, any double precision floating point number is represented as:

N = M * 2 (E-128)

where

$-1 \leq M < -1/2$ or $1/2 \leq M < 1$
$-32768 \leq E \leq 32767$.

Zero is represented as M = 0, E = 0

The value range, in octal (word1, word2, word3, word4) is:

('100000, '000000, '000000. '077777) [See Note] to
('077777, '177777, '177777, '077777)

corresponding to -1*2**32639 and (1-e)*2 32639

The values closest to zero, in octal, are:

('137777, '177777, '177777, '100000) and
('040000, '000000, '000000, '100000) [See Note]

corresponding to (-1/2+e)*2**-32896 and 1/2*2**-32896

Normalization ensures that bits 1 and 2 are different and is achieved by shifting left 1 bit at a time. Hence, the effective precision is between 46 and 47 bits.

Note

These numbers will cause exponent overflows if negated due to the asymmetry of 2's complement notation.

COMPLEX 64 bits. A complex number is defined as two REAL*4 entities (see above) representing the real and imaginary parts.

COMPLEX*16 128 bits. Same as COMPLEX, except that two DOUBLE PRECISION entities are used.

CHARACTERS Prime uses ASCII as its standard internal and external character code. It is the 8-bit, marking variety (parity bit always on). Thus, Prime's code set is effectively a 128-character code set. (ASCII spacing representation, parity bit always off, can be entered into the system, but most system software will fail to recognize the characters as their terminal printing equivalent.)

Each character occupies one byte. The length of a CHARACTER item may be up to 32767 characters.

APPENDIX D

ASCII CHARACTER SET

The standard character set used by Prime is the ANSI, ASCII 7-bit set.

PRIME USAGE

Prime hardware and software uses standard ASCII for communications with devices. The following points are particularly important to Prime usage.

- Output Parity is normally transmitted as a zero (space) unless the device requires otherwise, in which case software will compute transmitted parity. Some controllers (e.g., MLC) may have hardware to assist in parity generations.
- Input Parity is ignored by hardware and by standard software. Input drivers are responsible for making the parity bit suit the host software requirements. Some controllers (e.g., MLC) may assist in parity error detection.
- The Prime internal standard for the parity bit is one, i.e., '2ØØ is added to the octal value.

KEYBOARD INPUT

Non-printing characters may be entered into text with the logical escape character ^ and the octal value. The character is interpreted by output devices according to their hardware.

Example: Typing ^2Ø7 will enter one character into the text.

CTRL-P	('22Ø)	is interpreted as a .BREAK.
.CR.	('215)	is interpreted as a newline (.NL.)
"	('242)	is interpreted as a character erase
?	('277)	is interpreted as line kill
\	('334)	is interpreted as a logical tab (Editor)

Table D-1

ASCII Character Set (Non-Printing)

Octal Value	ASCII Char	Comments/Prime Usage	Control Char
2ØØ	NULL	Null character - filler	^@
2Ø1	SOH	Start of header (communications)	^A
2Ø2	STX	Start of text (communications)	^B
2Ø3	ETX	End of text communications	^C
2Ø4	EOT	End of transmission (communications)	^D
2Ø5	ENQ	End of I.D. (communications)	^E
2Ø6	ACK	Acknowledge affirmative (communications)	^F
2Ø7	BEL	Audible alarm (bell)	^G
21Ø	BS	Back space one position (carriage control)	^H
211	HT	Physical horizontal tab	^I
212	LF	Line feed; ignored as terminal input	^J
213	VT	Physical vertical tab (carriage control)	^K
214	FF	Form feed (carriage control)	^L
215	CR	Carriage return (carriage control) (1)	^M
216	SO	RRS-red ribbon shift	^N
217	SI	BRS-black ribbon shift	^O
22Ø	DLE	RCP-relative copy (2)	^P
221	DC1	RHT-relative horizontal tab (3)	^Q
222	DC2	HLF-half line feed forward (carriage control)	^R
223	DC3	RVT-relative vertical tab (4)	^S
224	DC4	HLR-half line feed reverse (carriage control)	^T
225	NAK	Negative acknowledgement (communications)	^U
226	SYN	Synchronocity (communications)	^V
227	ETB	End of transmission block (communications)	^W
23Ø	CAN	Cancel	^X
231	EM	End of Medium	^Y
232	SUB	Substitute	^Z
233	ESC	Escape	^[
234	FS	File separator	^\
235	GS	Group separator	^]
236	RS	Record separator	^^
237	US	Unit separator	^_

Notes for Table D-1

1. Interpreted as .NL. at the terminal.

2. .BREAK. at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceding line.

3. Next byte specifies number of spaces to insert.

4. Next byte specifies number of lines to insert.

Conforms to ANSI X3.4-1968

The parity bit ('2ØØ) has been added for Prime-usage.

Non-printing characters (^c) can be entered at most terminals by typing the (control) key and the c character key simultaneously.

Table D-2

ASCII Character Set (Printing)

Octal Value	ASCII Character	OCTAL Value	ASCII CHaracter	OCTAL Value	ASCII Character
240	.SP (1)	300	@	340	` (9)
241	!	301	A	341	a
242	" (2)	302	B	342	b
243	# (3)	303	C	343	c
244	$	304	D	344	d
245	%	305	E	345	e
246	&	306	F	346	f
247	' (4)	307	G	347	g
250	(	310	H	350	h
251	)	311	I	351	i
252	*	312	J	352	j
253	+	313	K	353	k
254	, (5)	314	L	354	l
255	-	315	M	355	m
256	.	316	N	356	n
257	/	317	O	357	o
260	0	320	P	360	p
261	1	321	Q	361	q
262	2	322	R	362	r
263	3	323	S	363	s
264	4	324	T	364	t
265	5	325	U	365	u
266	6	326	V	366	v
267	7	327	W	367	w
270	8	330	X	370	x
271	9	331	Y	371	y
272	:	332	Z	372	z
273	;	333	[	373	{
274	<	334	\	374	|
275	=	335	]	375	}
276	>	336	^ (7)	376	~ (10)
277	? (6)	337	_ (8)	377	DEL (11)

Notes for Table D-2

1. Space forward one position
2. Terminal usage - erase previous character
3. £ in British use
4. Apostrophe/single quote
5. Comma
6. Terminal usage - kill line
7. 1963 standard ↑; terminal use - logical escape
8. 1963 standard ←
9. Grave
10. 1963 standard ESC
11. Rubout - ignored

Conforms to ANSI X3.4-1968
1963 variances are noted

The parity bit ('200) has been added for Prime usage.